,

VŠB - Technická univerzita Ostrava

Fakulta strojní

Katedra aplikované mechaniky

Diplomová práce

Analýza problému rovinné pružnosti s pomocí adaptivních technik

Analysis of Two-Dimensional Linear Elasticity Problems using Adaptive Techniques

Student: Bc. Tomáš Ševčík

Vedoucí práce: Ing. Alexandros Markopoulos, Ph.D.

Ostrava 2017

VŠB - Technická univerzita Ostrava
Fakulta strojní
Katedra aplikované mechaniky

Zadání diplomové práce

Student: **Bc. Tomáš Ševčík**

Studijní program: N2301 Strojní inženýrství

Studijní obor: 3901T003 Aplikovaná mechanika

Téma: Analýza problémů rovinné pružnosti s pomoci adaptivních technik
Analysis of Two-Dimensional Linear Elasticity Problems using Adaptive Techniques

Jazyk vypracování: čeština

Zásady pro vypracování:

1. Popis používaných typu adaptivních metod a jejich využití při řešení inženýrských úloh.
2. Aplikace adaptivních metod ve vybraném komerčním programu (ANSYS, COMSOL, atd.).
3. Návrh a implementace vlastní konečnoprvkové knihovny s podporou adaptivních metod pro řešení úloh rovinné pružnosti.
4. Testování adaptivních metod na vhodném inženýrském problému se známým analytickým řešením. Provedení testů ve vlastním a komerčním software.

Seznam doporučené odborné literatury:

1. M. Asghar Bhatti: Fundamental Finite Element Analysis and Applications: with Mathematica and Matlab Computations, ISBN: 978-0-471-64808-6
1. M. Asghar Bhatti: Advanced Topics in Finite Element Analysis of Structures: With Mathematica and MATLAB Computations, ISBN: 978-0-471-64807-9
2. P. Solin: Partial Differential Equations and the Finite Element Method, J. Wiley & Sons, 2005

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Alexandros Markopoulos, Ph.D.**

Datum zadání: 09.12.2016

Datum odevzdání: 15.05.2017

doc. Ing. Radim Halama, Ph.D.
vedoucí katedry

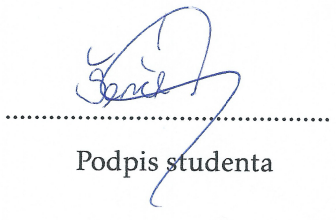
doc. Ing. Ivo Hlavatý, Ph.D.
děkan fakulty

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA FAKULTA STROJNÍ 3 OSTRAVA

Místopřísežné prohlášení studenta

Prohlašuji, že jsem celou diplomovou práci včetně příloh vypracoval samostatně pod vedením vedoucího diplomové práce a uvedl všechny použité podklady a literaturu.

V Ostravě dne 15. května 2017



.....

Podpis studenta

Prohlašuji, že:

- jsem si vědom, že na tuto moji závěrečnou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. Zákon o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (dále jen Autorský zákon), zejména § 35 (Užití díla v rámci občanských či náboženských obřadů nebo v rámci úředních akcí pořádaných orgány veřejné správy, v rámci školních představení a užití díla školního) a § 60 (Školní dílo),
- beru na vědomí, že Vysoká škola báňská – Technická univerzita Ostrava (dále jen „VŠB-TUO“) má právo užít tuto závěrečnou diplomovou práci nekomerčně ke své vnitřní potřebě (§ 35 odst. 3 Autorského zákona),
- bude-li požadováno, jeden výtisk této diplomové práce bude uložen u vedoucího práce,
- s VŠB-TUO, v případě zájmu z její strany, uzavřu licenční smlouvu s oprávněním užít dílo v rozsahu § 12 odst. 4 Autorského zákona,
- užít toto své dílo, nebo poskytnout licenci k jejímu využití, mohu jen se souhlasem VŠB-TUO, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly VŠB-TUO na vytvoření díla vynaloženy (až do jejich skutečné výše),
- beru na vědomí, že - podle zákona č 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů - že tato diplomová práce bude před obhajobou zveřejněna na pracovišti vedoucího práce, a v elektronické podobě uložena a po obhajobě zveřejněna v Ústřední knihovně VŠB-TUO, a to bez ohledu na výsledek její obhajoby.

V Ostravě dne 15. května 2017

.....

Podpis studenta

Jméno a příjmení autora práce: Tomáš Ševčík

Adresa trvalého pobytu autora práce: U Zahrádek 476, Slavičín

Anotace diplomové práce

ŠEVČÍK, T. *Analýza problému rovinné pružnosti s pomocí adaptivních technik: diplomová práce*. Ostrava: VŠB - Technická univerzita Ostrava, Fakulta strojní, Katedra aplikované mechaniky, 2017, 133 stran. Vedoucí práce: Markopoulos, A.

Diplomová práce se zabývá adaptivními technikami v metodě konečných prvků a jejich implementací ve vlastním konečno-prvkovém programu, jež slouží k řešení dvoudimenzionálních úloh. První část práce je věnována teoretické části, kde jsou zmíněny základní vztahy matematické teorie pružnosti, na ně navazuje teorie metody konečných prvků, načež jsou popsány principy adaptivních technik. Veškerá teorie byla poté uplatněna k implementaci vlastního konečno-prvkového algoritmu v prostředí programovacího jazyka *Python*. Na závěr je provedena komparace výsledků získaných vlastním MKP algoritmem a vybranými komerčními MKP programy.

Annotation of Master Thesis

ŠEVČÍK, T. *Analysis of Two-Dimensional Linear Elasticity Problems using Adaptive Techniques: Master Thesis*. Ostrava: VŠB - Technical University of Ostrava, Faculty of Mechanical Engineering, Department of Applied Mechanics, 2017, 133 pages. Head of Thesis: Markopoulos, A.

This thesis deals with adaptive techniques used in the finite element method (FEM). The main goal is to create the own algorithm for solving two-dimensional problems with adaptive techniques support. The first part is mainly focused on the theory. There are mentioned basic equations of the theory of elasticity. Significant part of the theory is dedicated to the FEM and adaptive techniques. The second part deals with the implementation of own adaptive algorithms in the programming languange *Python*. The last part presents comparsion of the own adaptive algorithms with commercial softwares *ANSYS APDL* and *Marc Mentat*.

Obsah

Seznam obrázků

Seznam tabulek

Seznam použitého značení

Latinská abeceda

Značka	Popis veličiny	Jednotka
$\boldsymbol{A}$	Matice koeficientů	[-]
$\boldsymbol{B}$	Matice derivací tvarových funkcí	[-]
c	Konstanta pro přesun středového uzlu	[-]
$\boldsymbol{D}$	Materiálová matice tuhosti	[-]
$\boldsymbol{D}^{-1}$	Materiálová matice poddajnosti	[-]
$\boldsymbol{D_D}$	Materiálová matice tuhosti pro stav rovinné deformace	[-]
$\boldsymbol{D_D}^{-1}$	Materiálová matice poddajnosti pro stav rovinné deformace	[-]
$\boldsymbol{D_N}$	Materiálová matice tuhosti pro stav rovinné napjatosti	[-]
$\boldsymbol{D_N}^{-1}$	Materiálová matice poddajnosti pro stav rovinné napjatosti	[-]
E	Materiálový modul pružnosti v tahu	[Pa]
E_e	Střední kvadratická chyba nad elementem	[-]
$\boldsymbol{e}_u$	Vektor chyby řešení stanovený z posuvů	[-]
$\boldsymbol{e}_\varepsilon$	Vektor chyby řešení stanovený z poměrných deformací	[-]
$\boldsymbol{e}_\sigma$	Vektor chyby řešení stanovený z napětí	[-]
$\boldsymbol{F}$	Globální vektor pravé strany	[-]
$F_{k,i}$	Silové účinky působící ve směru k, kde $k = x, y, z$	[N]
$\boldsymbol{f}$	Vektor osamělých sil	[-]
$\boldsymbol{f_p}$	Vektor plošných sil	[-]
$\boldsymbol{f_v}$	Lokální vektor pravé strany	[-]
$\boldsymbol{f_x}$	Vektor objemových sil	[-]
G	Materiálový modul pružnosti ve smyku	[Pa]
$\boldsymbol{I}$	Jednotková matice	[-]
$\boldsymbol{I}_k$	Vektor nesoucí informace o pozici řádků jednotlivých hodnot matice tuhosti $\boldsymbol{K}$	[-]
$\boldsymbol{J}$	Matice Jakobiánu transformace	[-]
$\boldsymbol{J}_k$	Vektor nesoucí informace o pozici sloupců jednotlivých hodnot matice tuhosti $\boldsymbol{K}$	[-]

Značka	Popis veličiny	Jednotka
$\boldsymbol{k}$	Lokální matice tuhosti elementu	[-]
$\boldsymbol{K}$	Globální matice tuhosti řešené úlohy	[-]
$\boldsymbol{K}_{k,j}$	Submatice globální matice tuhosti $\boldsymbol{K}$, kde $k, j = N, \Gamma$	[-]
$\boldsymbol{L}$	Dolní trojúhelníková matice	[-]
$\boldsymbol{M}$	Matice bázových funkcí	[-]
$M_{k,i}$	Momentové účinky kolem osy k, kde $k = x, y, z$	[Nm]
m	Počet integračních bodů	[-]
$\boldsymbol{N}$	Matice tvarových funkcí	[-]
$\tilde{\boldsymbol{N}}$	Matice tvarových funkcí pro interpolaci geometrie elementu	[-]
n	Počet stupňů volnosti elementu	[-]
n_d	Počet členů nutný k sestavení polynomu r-tého stupně	[-]
$\boldsymbol{P}$	Vektor souřadnic uzlu před přesunem	[-]
$\boldsymbol{P}_{new}$	Vektor souřadnic přesunutého uzlu	[-]
$p_{0,1}$	Tlak působící na vnitřní/vnější plášť tlustostěnné nádoby	[Pa]
$\boldsymbol{p}$	Vektor povrchových sil	[-]
$R_{0,1}$	Vnitřní/vnější poloměr tlustostěnné nádoby	[m]
r	Stupeň aproximačního polynomu	[-]
$\boldsymbol{r}_\sigma$	Vektor uzlových napětí	[-]
t	Tloušťka elementu	[m]
$\boldsymbol{U}$	Horní trojúhelníková matice	[-]
$\boldsymbol{u}$	Vektor posuvů	[-]
u	Posuv ve směru osy x	[m]
$\hat{\boldsymbol{u}}$	Vektor aproximovaných posuvů	[-]
$\boldsymbol{V}_k$	Vektor obsahující hodnoty matice tuhosti $\boldsymbol{K}$	[-]
v	Posuv ve směru osy y	[m]
$\boldsymbol{W}$	Matice hmotnosti	[-]
w	Posuv ve směru osy z	[m]
X	Objemová síla působící ve směru osy x	[Nm^{-3}]
x_T	X-ová souřadnice těžiště	[m]

Značka	Popis veličiny	Jednotka
Y	Objemová síla působící ve směru osy y	[Nm^{-3}]
y_T	Y-ová souřadnice těžiště	[m]
Z	Objemová síla působící ve směru osy z	[Nm^{-3}]

Řecká abeceda

Značka	Popis veličiny	Jednotka
$\boldsymbol{\alpha}$	Vektor obecných souřadnic	[-]
$\gamma_{j,k}$	Zkos v příslušné rovině, kde $j,k = x,y,z$	[-]
Δu	Střední kvadratická chyba posuvů	[-]
$\Delta\sigma$	Střední kvadratická chyba napětí	[-]
ε_k	Poměrné prodloužení v příslušném směru, kde $k = x,y,z$	[-]
$\boldsymbol{\varepsilon}$	Vektor složek tenzoru poměrné deformace	[-]
η	Přirozená souřadnice v bezrozměrném prostoru	[-]
μ	Poissonovo číslo	[-]
ξ	Přirozená souřadnice v bezrozměrném prostoru	[-]
σ_k	Normálové napětí v příslušném směru, kde $k = x,y,z$	[Pa]
$\boldsymbol{\sigma}$	Vektor složek tenzoru napětí	[-]
$\hat{\boldsymbol{\sigma}}$	Vektor aproximovaných napětí	[-]
$\boldsymbol{\sigma}^*$	Vektor vyhlazeného napětí	[-]
Φ	Váhový koeficient	[-]
ψ	Relativní procentuální chyba	[%]
$\bar{\psi}$	Přípustná relativní procentuální chyba	[%]

1 Úvod

Automatizace, pojem který byl počátkem šedesátých let dvacátého století pouze science fiction, je dnes již součástí každodenního života. Díky rozvoji počítačových technologií se automatizace stala cestou, jak urychlit, zdokonalit nebo dokonce zcela nahradit lidskou činnost. V případě metody konečných prvků (MKP) je stále zapotřebí lidské obsluhy, která vytvoří nebo upraví model součásti, provede diskretizaci, dohledá správné materiálové parametry a aplikuje patřičné okrajové či počáteční podmínky. Řešení však může být kvůli nevhodně zvolené konečno-prvkové síti zatíženo značnou diskretizační chybou. Z toho důvodu se musí síť vhodně upravit a následně provést nový výpočet. Tento postup se poté může mnohokrát opakovat než se dosáhne požadované přesnosti řešení. Pro zvýšení rychlosti a efektivity návrhu optimálnější sítě lze využít procesu automatizace, který v metodě konečných prvků nese název *adaptivní technika*.

Algoritmizace adaptivních technik se začala rozvíjet s nástupem aposteriorních odhadů chyby řešení. Nejvíce používaným postupem pro aposteriorní odhad chyby se stala metoda autorů Zienkiewicze a Zhua. Metoda je založena na principu stanovení energetické chyby na elementu v důsledku nespojitosti napětí či deformací mezi elementy. Vhodným normováním energetické chyby lze následně detekovat elementy, které nesplňují stanovená kritéria přesnosti a jsou tedy ideálními adepty k přesíťování. Tento postup se opakuje dokud na všech elementech neklesne hodnota chyby pod stanovenou úroveň nebo dokud se nedosáhne maximálního počtu iterací.

Adaptivní techniky založeny na principu metody Zienkiewicze a Zhua jsou součástí komerčních programů jako je kupříkladu *Marc Mentat* či *ANSYS APDL / Workbench*.

Problematika adaptivních technik je podrobně popsána v kapitole 5, kde jsou taktéž detailněji vylíčeny principy adaptace sítě výše zmíněných komerčních programů.

2 Cíle práce

Cílem diplomové práce je vytvořit vlastní MKP algoritmus pro řešení dvoudimenzionálních úloh včetně podpory adaptivních technik. Práce je standardně rozdělena do dvou částí.

První část je věnována převážně teorii. Nejprve jsou popsány elementární vztahy teorie pružnosti a specifikovány dva základní stavy rovinné úlohy. Následně je definován princip metody konečných prvků včetně přiblížení teorie referenčních prvků, které byly využity v implementovaném konečno-prvkovém algoritmu. Na závěr teoretické části je uvedena problematika adaptivních technik se zaměřením na aposteriorní odhad chyby metodou Zienkiewicze a Zhua.

V druhé části práce je využíváno teoretických poznatků při implementaci vlastního konečno-prvkového algoritmu pro řešení dvoudimenzionálních problémů s podporou adaptace sítě. Princip algoritmu je popsán na úloze tlustostěnné nádoby, pro který byl také primárně vytvořen (určité funkce jsou dostupné pouze pro případ tlustostěnné nádoby). Jelikož může vést adaptace sítě k poměrně značnému nárůstu počtu elementů, je kromě přímého řešiče (*LU transformace*) implementován také iterační řešič (*metoda sdružených gradientů*). Poslední částí práce je validace vlastního MKP algoritmu se zvolenými komerčními konečno-prvkovými programy na vybraných dvourozměrných úlohách.

3 Základní vztahy matematické teorie pružnosti

Uvedené vztahy a formy zápisu jednotlivých veličin v této kapitole byly převzaty z (1).Díky matematické teorii pružnosti lze určit na obecném tělese o objemu V ohraničeném hranicí Γ tři pole:

- vektor posuvů $\mathbf{u} = \{u, v, w\}^T$,
- vektor složek tenzoru poměrné deformace $\boldsymbol{\varepsilon} = \left\{\varepsilon_x, \varepsilon_y, \varepsilon_z, \gamma_{yz}, \gamma_{zx}, \gamma_{xy}\right\}^T$,
- vektor složek tenzoru napětí $\boldsymbol{\sigma} = \left\{\sigma_x, \sigma_y, \sigma_z, \tau_{yz}, \tau_{zx}, \tau_{xy}\right\}^T$.

Aby úloha o 15 neznámých mohla být vyřešena, je zapotřebí systému 15 rovnic:

- 6 geometrických rovnic,
- 3 rovnice rovnováhy,
- 6 fyzikálních rovnic.

3.1 Geometrické rovnice

Pomocí geometrických rovnic lze popsat závislost poměrných deformací na posuvech. Existují dva základní přístupy pro odvození těchto rovnic. Jednodušší způsob, který byl odvozen na základě předpokladu malých deformací, definoval v roce 1823 A. L. Cauchy. Z tohoto důvodu platí Cauchyho rovnice pouze v případě malých deformací.

$$\begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ \gamma_{yz} \\ \gamma_{zx} \\ \gamma_{xy} \end{Bmatrix} = \begin{bmatrix} \frac{\partial u}{\partial x} & 0 & 0 \\ 0 & \frac{\partial v}{\partial y} & 0 \\ 0 & 0 & \frac{\partial w}{\partial z} \\ 0 & \frac{\partial v}{\partial z} & \frac{\partial w}{\partial y} \\ \frac{\partial u}{\partial z} & 0 & \frac{\partial w}{\partial x} \\ \frac{\partial u}{\partial y} & \frac{\partial v}{\partial x} & 0 \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} & 0 \\ 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial x} & 0 & \frac{\partial}{\partial z} \\ 0 & 0 & \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{bmatrix}^T \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} = \partial^T \boldsymbol{u} \quad (1)$$

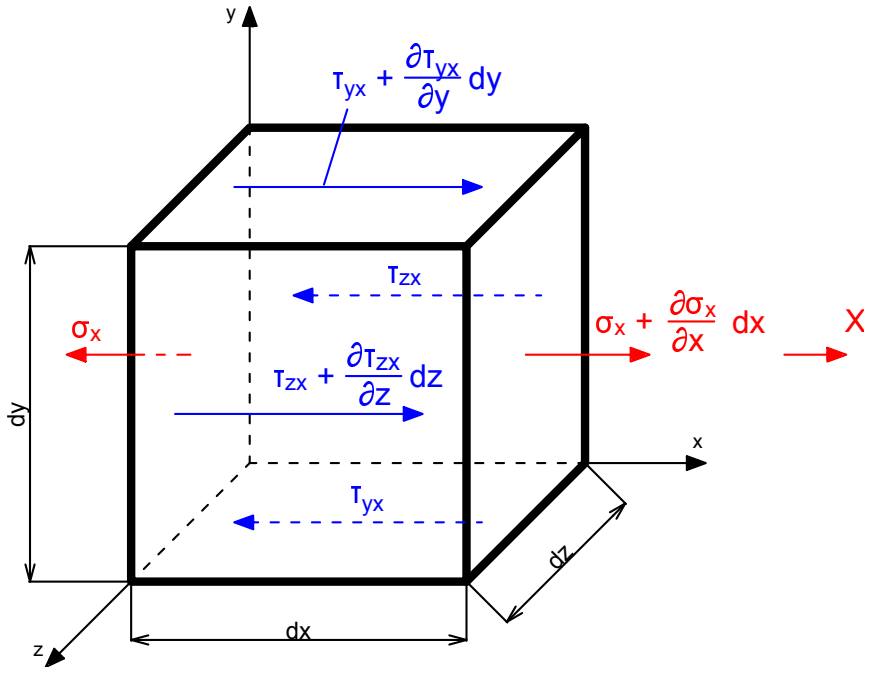
Šest složek deformace (přetvoření) je zde vyjádřeno pomocí tří posuvů. Z toho vyplývá, že složky deformace nejsou zcela nezávislými funkcemi souřadnic. Pokud mají popisovat deformace spojitého tělesa, musí splňovat deformační podmínky, které jsou získány vyloučením posuvů z Cauchyho rovnic. Tímto postupem jsou zjištěny tzv. *rovnice kompatibility*.

3.2 Rovnice rovnováhy

Aby obecné těleso v prostoru bylo v rovnovážném stavu, musí být součet všech sil a momentů, působících na těleso, roven nule

$$\begin{aligned} \sum_{i=1}^{N} F_{x,i} = 0, \qquad & \sum_{i=1}^{N} M_{x,i} = 0, \\ \sum_{i=1}^{N} F_{y,i} = 0, \qquad & \sum_{i=1}^{N} M_{y,i} = 0, \\ \sum_{i=1}^{N} F_{z,i} = 0, \qquad & \sum_{i=1}^{N} M_{z,i} = 0. \end{aligned} \tag{2}$$

Jestliže je těleso v rovnováze, musí být v rovnováze i nekonečně malý element. Na obrázku 1 je pro přehlednost znázorněna pouze objemová síla X a také všechna napětí rovnoběžná s osou x.



Obrázek 1: Elementární krychle se znázorněnými složkami zatížení ve směru osy x.

Sestavením rovnic rovnováhy na nekonečně malém elementu lze určit diferenciální rovnice rovnováhy ve tvaru:

$$\begin{aligned}
\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{xz}}{\partial z} + X = 0, \\
\frac{\partial \tau_{yx}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{yz}}{\partial z} + Y = 0, \\
\frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{zy}}{\partial y} + \frac{\partial \sigma_z}{\partial z} + Z = 0.
\end{aligned} \tag{3}$$

Momentové podmínky rovnováhy jsou vždy splněny neboť popisují zákon o sdruženosti smykových napětí

$$\tau_{xy} = \tau_{yx}, \qquad \tau_{yz} = \tau_{zy}, \qquad \tau_{xz} = \tau_{zx}. \tag{4}$$

3.3 Fyzikální rovnice

Fyzikálními rovnicemi je vyjádřena závislost mezi napětími a deformacemi. Pro pružný, homogenní a izotropní materiál lze fyzikální rovnice interpretovat pomocí Hookova zákona

$$\begin{aligned}
\varepsilon_x = \frac{1}{E}\Big[\sigma_x - \mu\Big(\sigma_y + \sigma_z\Big)\Big], \\
\varepsilon_y = \frac{1}{E}\Big[\sigma_y - \mu\left(\sigma_x + \sigma_z\right)\Big], \\
\varepsilon_z = \frac{1}{E}\Big[\sigma_z - \mu\Big(\sigma_x + \sigma_y\Big)\Big], \\
\gamma_{yz} = \frac{\tau_{yz}}{G}, \\
\gamma_{zx} = \frac{\tau_{zx}}{G}, \\
\gamma_{xy} = \frac{\tau_{xy}}{G},
\end{aligned} \tag{5}$$

kde E představuje modul pružnosti v tahu, μ je Poissonovo číslo a G je modul pružnosti ve smyku. Mezi veličinami platí následující vztah

$$G = \frac{E}{2\left(1+\mu\right)}. \tag{6}$$

Nebo lze využít zkrácené formy zápisu

$$\boldsymbol{\varepsilon} = \boldsymbol{D}^{-1}\boldsymbol{\sigma}, \tag{7}$$

kde matice $\boldsymbol{D}$ je *materiálová matice tuhosti*. Pro izotropní materiál má následující tvar

$$\boldsymbol{D} = \frac{E}{(1+\mu)(1-2\mu)} \begin{bmatrix} (1-\mu) & \mu & \mu & 0 & 0 & 0 \\ \mu & (1-\mu) & \mu & 0 & 0 & 0 \\ \mu & \mu & (1-\mu) & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\mu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\mu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\mu}{2} \end{bmatrix} \tag{8}$$

a její inverzní matice $\boldsymbol{D}^{-1}$ je *materiálová matice poddajnosti*

$$\boldsymbol{D}^{-1} = \frac{1}{E} \begin{bmatrix} 1 & -\mu & -\mu & 0 & 0 & 0 \\ -\mu & 1 & -\mu & 0 & 0 & 0 \\ -\mu & -\mu & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2(1+\mu) & 0 & 0 \\ 0 & 0 & 0 & 0 & 2(1+\mu) & 0 \\ 0 & 0 & 0 & 0 & 0 & 2(1+\mu) \end{bmatrix}. \tag{9}$$

3.4 Okrajové podmínky

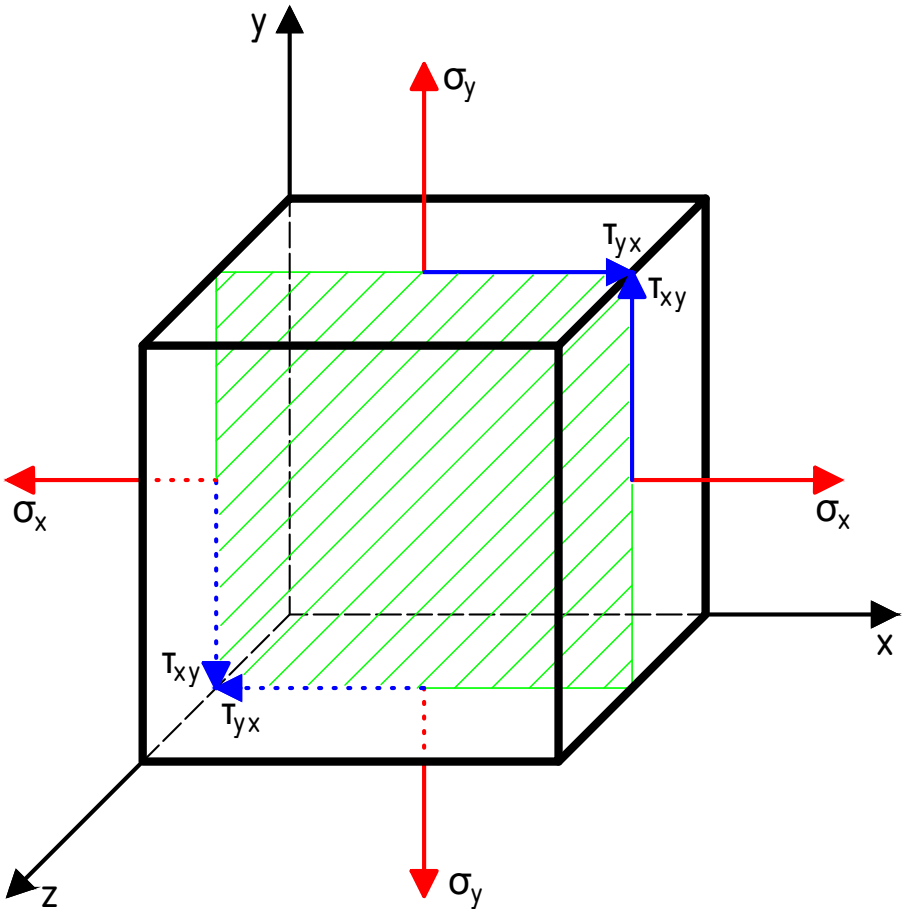
Pro stanovení napjatosti tělesa v každém jeho bodě musí platit takové řešení, aby napjatost v bodech tělesa na povrchu odpovídala předepsanému zatížení, popřípadě, aby byly v bodech na povrchu tělesa splněny předepsané posuvy. Tyto podmínky se označují jako *okrajové podmínky* a mohou být formulovány dvojím způsobem:

- **Statické - silové okrajové podmínky** - předepisují hodnotu složek vektoru zatížení na části povrchu (hranice) tělesa.
- **Geometrické - kinematické okrajové podmínky** - předepisují hodnotu složek vektoru posunutí na části povrchu (hranice) tělesa.

3.5 Rovinná úloha

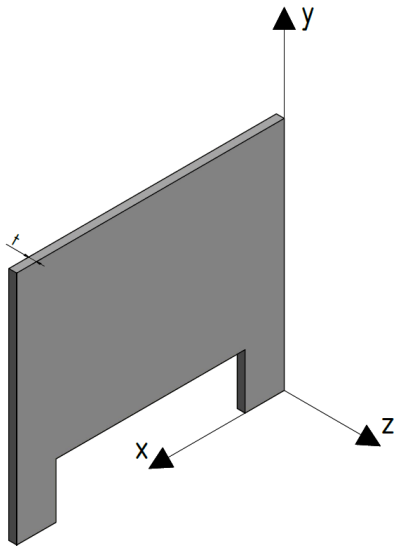
Řešení prostorových úloh může být poměrně obtížné, proto je značně výhodné využívat možnosti zjednodušení na dvourozměrnou případně jednorozměrnou úlohu. Jelikož se práce zabývá analýzou rovinné pružnosti, bude zde zmíněno pouze zjednodušení na problém rovinné úlohy.

Rovinná úloha je speciálním případem obecné prostorové úlohy, kde všechny nenulové složky napětí leží pouze v jedné rovině, jak je znázorněno na obrázku 2.



Obrázek 2: Rovinná úloha.

Jestliže je kromě úvodního předpokladu nulová složka tenzoru napětí σ_z, jedná se o případ *rovinné napjatosti*. Typickým případem rovinné napjatosti je deska konstantní tloušťky, která je zatížena pouze ve střednicové rovině, jak je znázorněno na obrázku 3.



Obrázek 3: Příklad úlohy rovinné napjatosti.

Tenzor poměrné deformace a tenzor napětí jsou využitím rovinné napjatosti zjednodušeny do podoby

$$\boldsymbol{\varepsilon} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \varepsilon_z \\ 0 \\ 0 \\ \gamma_{xy} \end{Bmatrix} \qquad \boldsymbol{\sigma} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ 0 \\ 0 \\ 0 \\ \tau_{xy} \end{Bmatrix}. \tag{10}$$

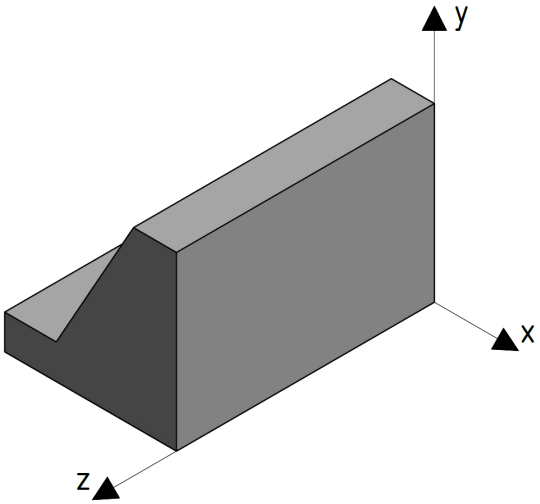
Využitím obecného Hookova zákona pro složku tenzoru poměrné deformace ε_z lze psát

$$\varepsilon_z = \frac{1}{E}\left[0 - \mu\left(\sigma_x + \sigma_y\right)\right]. \tag{11}$$

Pro rovinnou napjatost má matice tuhosti $\boldsymbol{D_N}$ tvar

$$\boldsymbol{D_N} = \frac{E}{1-\mu^2}\begin{bmatrix} 1 & \mu & 0 \\ \mu & 1 & 0 \\ 0 & 0 & \dfrac{1-\mu}{2} \end{bmatrix}. \tag{12}$$

Pokud kromě výše zmíněného předpokladu je složka tenzoru poměrné deformace ε_z rovna nule, jedná se o případ *rovinné deformace*. Praktický případ rovinné deformace je nejčastěji spojen s faktem, že rozměr ve směru osy z převažuje nad ostatními rozměry. Klasickým příkladem může být úloha přehradní zdi, jak je znázorněno na obrázku 4.



Obrázek 4: Příklad úlohy rovinné deformace.

Tenzor poměrné deformace a tenzor napětí jsou využitím rovinné deformace zjednodušeny do podoby

$$\boldsymbol{\varepsilon} = \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ 0 \\ 0 \\ 0 \\ \gamma_{xy} \end{Bmatrix} \qquad \boldsymbol{\sigma} = \begin{Bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \\ 0 \\ 0 \\ \tau_{xy} \end{Bmatrix}. \tag{13}$$

Opětovným užitím Hookova zákona pro složku tenzoru poměrné deformace ε_z, lze dokázat, že složka tenzoru napětí σ_z není rovna nule

$$\varepsilon_z = \frac{1}{E}\left[\sigma_z - \mu\left(\sigma_x + \sigma_y\right)\right] = 0. \tag{14}$$

Matice tuhosti pro rovinnou deformaci $\boldsymbol{D_D}$ má tvar

$$\boldsymbol{D_D} = \frac{E(1-\mu)}{(1+\mu)(1-2\mu)} \begin{bmatrix} 1 & \frac{\mu}{1-\mu} & 0 \\ \frac{\mu}{1-\mu} & 1 & 0 \\ 0 & 0 & \frac{1-2\mu}{2(1-\mu)} \end{bmatrix}. \tag{15}$$

Rovinný problém, to je rovinnou napjatost a rovinný stav deformace, lze formálně popsat stejnými fyzikálními rovnicemi. Toho lze docílit vhodnou úpravou matice poddajnosti pro případ rovinné deformace tak, že se nejprve provede vyjádření σ_z z rovnice 14

$$\sigma_z = \mu\left(\sigma_x + \sigma_y\right). \tag{16}$$

Do fyzikálních rovnic pro výpočet složek tenzoru deformace $\varepsilon_x, \varepsilon_y$ je následně dosazena rovnice 16

$$\begin{aligned} \varepsilon_x &= \frac{1-\mu^2}{E}\left[\sigma_x - \frac{\mu}{1-\mu}\sigma_y\right] \\ \varepsilon_y &= \frac{1-\mu^2}{E}\left[\sigma_y - \frac{\mu}{1-\mu}\sigma_x\right] \end{aligned} \tag{17}$$

Matice poddajnosti materiálu pro rovinný stav deformace má po úpravě tvar

$$\boldsymbol{D_D}^{-1} = \frac{1-\mu^2}{E}\begin{bmatrix} 1 & -\frac{\mu}{1-\mu} & 0 \\ -\frac{\mu}{1-\mu} & 1 & 0 \\ 0 & 0 & \frac{2}{1-\mu} \end{bmatrix}. \tag{18}$$

Zavedou-li se následující substituce

$$\begin{aligned} E_1 &= \frac{E}{1-\mu^2}, \\ \mu_1 &= \frac{\mu}{1-\mu}, \end{aligned} \tag{19}$$

a provede-li se zpětná inverze bude mít matice tuhosti pro rovinný stav deformace $\boldsymbol{D_D}$ formálně stejný zápis jako matice tuhosti pro rovinnou napjatost $\boldsymbol{D_N}$

$$\boldsymbol{D_D} = \frac{E_1}{1-\mu_1^2}\begin{bmatrix} 1 & \mu_1 & 0 \\ \mu_1 & 1 & 0 \\ 0 & 0 & \frac{1-\mu_1}{2} \end{bmatrix}. \tag{20}$$

4 Metoda konečných prvků

Historie metody konečných prvků se začala psát roku 1941, kdy Hrenninkoff publikoval svoji práci, zabývající se řešením rozsáhlých rámových konstrukcí pomocí metody s názvem *„Frame work method“*. Nejčastěji se však vznik metody konečných prvků připisuje matematiku Courantovi, který v roce 1943 publikoval práci, ve které navrhl koncept po částech spojitých funkcí nad danou podoblastí. Levy v roce 1947 rozvinul silovou metodu a v roce 1953 ve své práci poznamenal, že slibnou alternativní metodou pro analýzu leteckých konstrukcí by mohla být metoda deformační. Nicméně jeho rovnice byly pro manuální řešení velmi pracné, a proto se začala tato metoda používat až s nástupem počítačů.

V roce 1956 Turner a kolektiv provedl první analýzu rovinné napjatosti. Pro diskretizaci byly využity tří-uzlové elementy trojúhelníkového tvaru a byl nastíněn proces nazývaný *„direct stiffness method“* (deformační metoda), který spočívá v sestavení globální matice tuhosti řešené úlohy. Pojem *„finite element“* (konečný prvek) byl poprvé představen v roce 1960 Cloughem při analýze rovinné napjatosti s využitím trojúhelníkových a čtyřúhelníkových elementů.

V roce 1965 Melosh položil důkaz, že metoda konečných prvků je moderní variantou Ritzovy metody a lze ji využít pro řešení širokého spektra fyzikálních problémů. Další stěžejní práce pro metodu konečných prvků vytvořili autoři jako Zienkiewicz, Cheung, Gallagher, Marcalo, Oden, Veubeke, z našich autorů pak například Kolář, Kratochvíl, Zlámal a Ženíšek.

V dnešní době je metoda konečných prvků nedílnou součástí vývojových pracovišť, neboť dovoluje počítat velmi rozsáhlé a komplikované úlohy bez nutnosti aplikace různých zjednodušení.

Moderní programy zabývající se MKP jsou vytvářeny jako expertní systémy, které umožňují vytvářet modely, mnohdy samy vybírají vhodné schéma výpočtu a také umožňují zpětnou kontrolu přesnosti výsledků. *„Problémy Apolla 11 dokumentují, že programy s prvky inteligence a optimalizace jsou i při maximálním úsilí tvůrců náchylné k chybám (programátorský folklór praví, že každý program lze zkrátit na jediný příkaz, který je chybný)“*(7). Jedinou možností detekce chyby výpočtu je dokonalá znalost „zákulisí“ analyzovaných jevů včetně principů samotné výpočetní metody.

4.1 Variační metoda MKP

Variační metody v mechanice vycházejí z principu virtuálních prací. Dle formulace principu virtuálních prací jsou rozlišovány dva variační principy:

- **Lagrangeův variační princip** - odvozen z principu virtuálních posuvů,
- **Castiglianův variační princip** - odvozen z principu virtuálních sil.

Lagrangeův princip virtuálních posunutí byl odvozen variací potencionální energie systému dle složek posunutí a deformací, kdežto Castiglianův princip virtuálních sil vznikl variací komplementární potencionální energie systému dle silových složek (síly + napětí).
Interpretace metody konečných prvků musí splňovat tři základní požadavky:

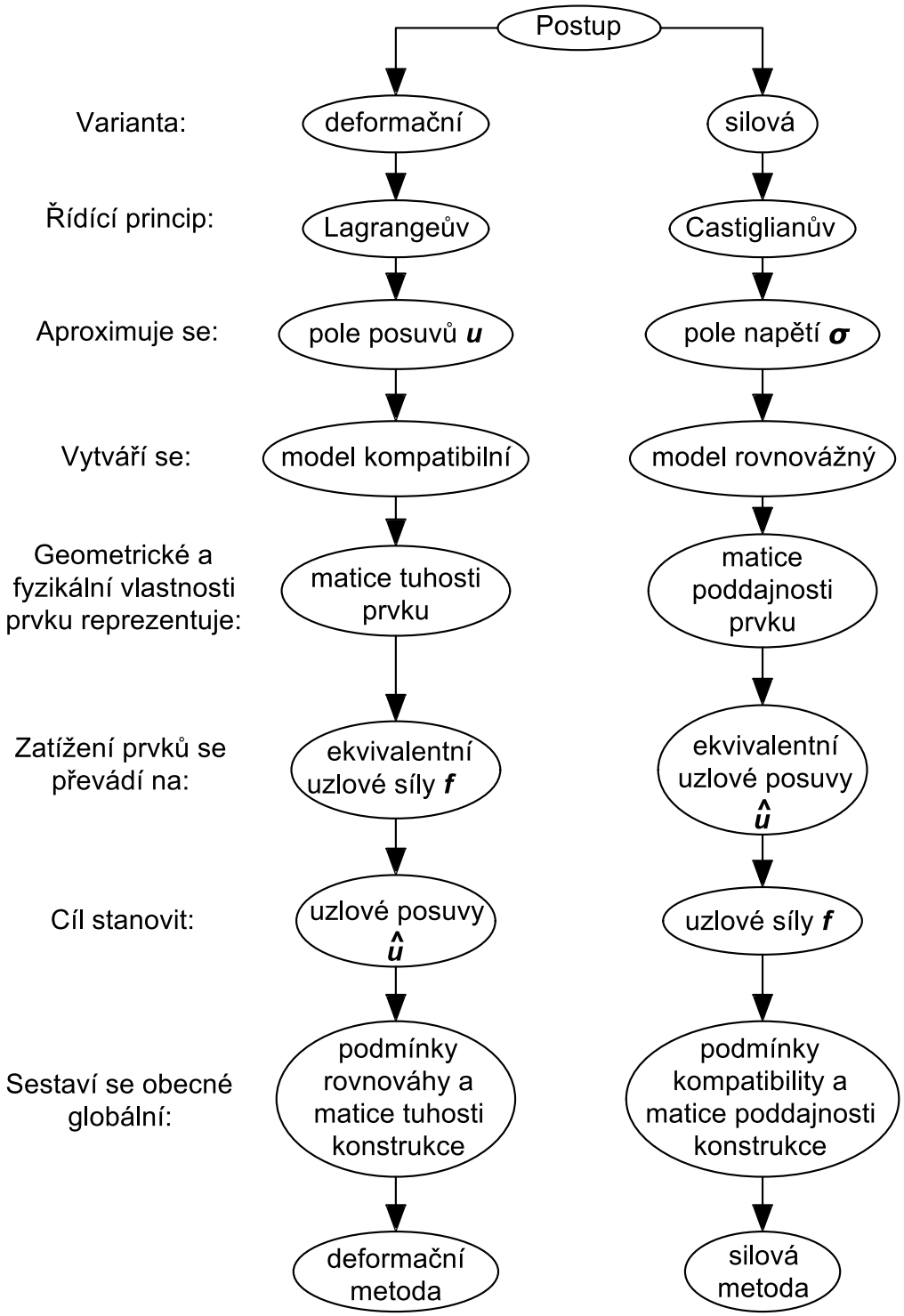
- rovnováhu tělesa jako celku včetně rovnováhy každého jeho prvku,
- vzájemnou kompatibilitu jednotlivých částí tělesa,
- platnost předpokládaných fyzikálních vztahů mezi tenzory napětí a deformace.

Aproximačními funkcemi lze obvykle splnit pouze jeden z požadavků rovnováhy či kompatibility. Jde poté o *model rovnovážný* nebo o *model kompatibilní*.

Kompatibilní model je spjat s deformační variantou metody konečných prvků, kterou je aproximováno pole posunutí $\boldsymbol{u}$ aproximačními funkcemi. Využitím aproximačních funkcí je následně stanoven funkcionál potencionální energie, jehož minimalizací je získána soustava rovnic pro stanovení neznámého pole posuvů $\hat{\boldsymbol{u}}$.

Rovnovážný model je naopak spojen se silovou variantou metody konečných prvků, kterou je aproximováno pole napětí σ aproximačními funkcemi.

Při správné volbě aproximačních funkcí (musí splňovat podmínky kompatibility) jsou u deformační varianty předem splněny rovnice rovnováhy. Naopak u silové metody jsou apriori splněny podmínky kompatibility. Metodika obou zmíněných variant je zřejmá z obrázku 5.



Obrázek 5: Postup při využití deformační či silové varianty (7).

Značnou částí komerčních MKP programů je používána právě deformační metoda s využitím kompatibilního modelu. Popularita této metody je způsobena menším počtem hledaných veličin než v případě metody silové. Proto byla zvolena i pro implementaci vlastního algoritmu.

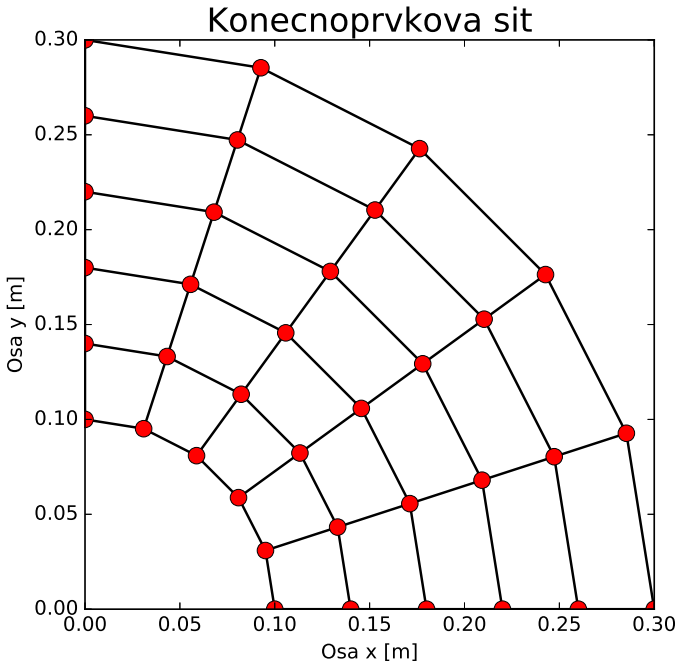
4.2 Princip metody konečných prvků

V následující kapitole bude popsán postup řešení metodou konečných prvků pro rovinnou úlohu. Jelikož je odvození vztahů zmíněno v mnoha knihách, budou zde uvedeny pouze koncové vzorce. Jejich podrobné odvození, včetně výše zmíněných principů virtuálních prací a variačních principů, je velmi názorně uvedeno v literatuře (5) a (7).

Postup řešení metodou konečných prvků lze rozdělit na několik základních kroků:

a) Rozdělení zkoumaného tělesa na prvky

Pro rovinnou úlohu jsou nejčastěji využívány elementy trojúhelníkového tvaru, které jsou výhodné při síťovaní komplikovaných tvarů. Pokud je zkoumané těleso tvarově jednoduché, je lepší využít čtyřúhelníkových elementů. Elementy s větším počtem hran než čtyři nejsou prakticky vůbec používány. Samotné rozdělení zkoumaného tělesa na konečný počet elementů je do značné míry věcí intuice a může značně ovlivnit přesnost řešení. Na obrázku 6 je vidět diskretizace tlustostěnné nádoby na konečný počet (25) čtyř-uzlových elementů.



Obrázek 6: Rozdělení tlustostěnné nádoby na 25 čtyřúhelníkových čtyř-uzlových elementů.

b) Volba stupně aproximačního polynomu tvarových funkcí

Dalším důležitým krokem je stanovení velikosti stupně aproximačního polynomu tvarových funkcí $\boldsymbol{N}$, jež se využívají na aproximaci pole posunutí $\boldsymbol{u}$

$$\boldsymbol{u} = \boldsymbol{N}\,\hat{\boldsymbol{u}}. \tag{21}$$

c) Stanovení lokální matice tuhosti jednotlivých prvků

Každý jednotlivý element vzdoruje deformaci na základě svých geometrických a fyzikálních vlastností. Tato schopnost elementu je vyjádřena maticí tuhosti, jejíž odvození je základním kamenem metody konečných prvků.

Z principu virtuálních posunutí lze odvodit vztah pro výpočet lokální matice tuhosti $\boldsymbol{k}$

$$\boldsymbol{k} = \int_{\Omega} \boldsymbol{B}^T \boldsymbol{D}\,\boldsymbol{B}\; t\; d\Omega, \tag{22}$$

kde matice $\boldsymbol{B}$ vznikne součinem transponované operátorové matice pro rovinnou úlohu ∂^T a matice tvarových funkcí $\boldsymbol{N}$

$$\boldsymbol{B} = \begin{bmatrix} \dfrac{\partial}{\partial x} & 0 & 0 & \dfrac{\partial}{\partial y} \\ 0 & \dfrac{\partial}{\partial y} & 0 & \dfrac{\partial}{\partial x} \end{bmatrix}^T \boldsymbol{N} = \partial^T \boldsymbol{N}. \tag{23}$$

Matice tuhosti závisí také na tloušťce elementu t.

d) Stanovení lokálních příspěvků pravé strany

Na každý element mohou působit různé druhy vnějších zatížení, které se poté transformují do příslušných uzlů elementu a vytvářejí lokální vektor pravé strany

$$\boldsymbol{f_v} = \boldsymbol{f} + \boldsymbol{f_p} + \boldsymbol{f_x}, \tag{24}$$

kde symbolem $\boldsymbol{f}$ je označen vektor osamělých sil, $\boldsymbol{f_p}$ představuje vektor plošných sil, jehož výpočet je dán vztahem

$$\boldsymbol{f_p} = \int_{\Gamma_p} \boldsymbol{N}^T \boldsymbol{p}\; t\; dL \tag{25}$$

a $\boldsymbol{f_x}$ označuje vektor objemových sil, který je dán vztahem

$$\boldsymbol{f_x} = \int_{V} \boldsymbol{N}^T \boldsymbol{X}\; t\; dV. \tag{26}$$

Vektor $\boldsymbol{p}$ odpovídá vektoru povrchových sil, působících na „hranici“ elementu $\boldsymbol{\Gamma_p}$ a symbolem $\boldsymbol{X}$ je interpretováno objemové zatížení, působící na „objem“ elementu dV.

e) Opětovné složení jednotlivých prvků

Jedná se o začlenění vlivu každého jednotlivého prvku jak do globální (celkové) matice tuhosti $\boldsymbol{K}$, tak i do globálního vektoru pravé strany $\boldsymbol{F}$ a následného sestavení soustavy rovnic, jež odpovídají rovnováze vnitřních a vnějších sil

$$\boldsymbol{K}\ \hat{\boldsymbol{u}} = \boldsymbol{F}. \tag{27}$$

Řešením této soustavy rovnic je vektor neznámých zobecněných posuvů v uzlech $\hat{\boldsymbol{u}}$.

f) Aplikace okrajových podmínek

Pokud by byla výše zmíněná soustava rovnic řešena bez aplikace okrajových podmínek, úloha by měla nekonečně mnoho řešení. Jinými slovy, globální matice tuhosti $\boldsymbol{K}$ by byla singulární (determinant globální matice tuhosti $\boldsymbol{K}$ by byl roven nule). Pro řešení statických úloh pružnosti s využitím deformační varianty MKP lze definovat jednoduchou poučku: Při řešení musí být předepsány alespoň takové okrajové podmínky, aby zamezily pohybu tělesa jako celku ve všech směrech, které jsou pro daný typ a dimenzi úlohy přípustné.

g) Dopočet sekundárních neznámých

Jakmile je vyřešena soustava rovnic 27, je možno na základě stanovených uzlových posuvů dopočítat sekundární veličiny jako jsou poměrné deformace $\boldsymbol{\varepsilon}$ či napětí $\boldsymbol{\sigma}$.

Pro stanovení vektoru poměrných deformací na elementu $\boldsymbol{\varepsilon}^e$ lze psát

$$\boldsymbol{\varepsilon}^e = \partial^T\ \boldsymbol{N}\ \hat{\boldsymbol{u}}^e = \boldsymbol{B}\ \hat{\boldsymbol{u}}^e \tag{28}$$

a s využitím Hookova zákona je možno stanovit napětí na elementu $\boldsymbol{\sigma}^e$ jako

$$\boldsymbol{\sigma}^e = \boldsymbol{D}\ \boldsymbol{\varepsilon}^e = \boldsymbol{D}\ \boldsymbol{B}\ \hat{\boldsymbol{u}}^e. \tag{29}$$

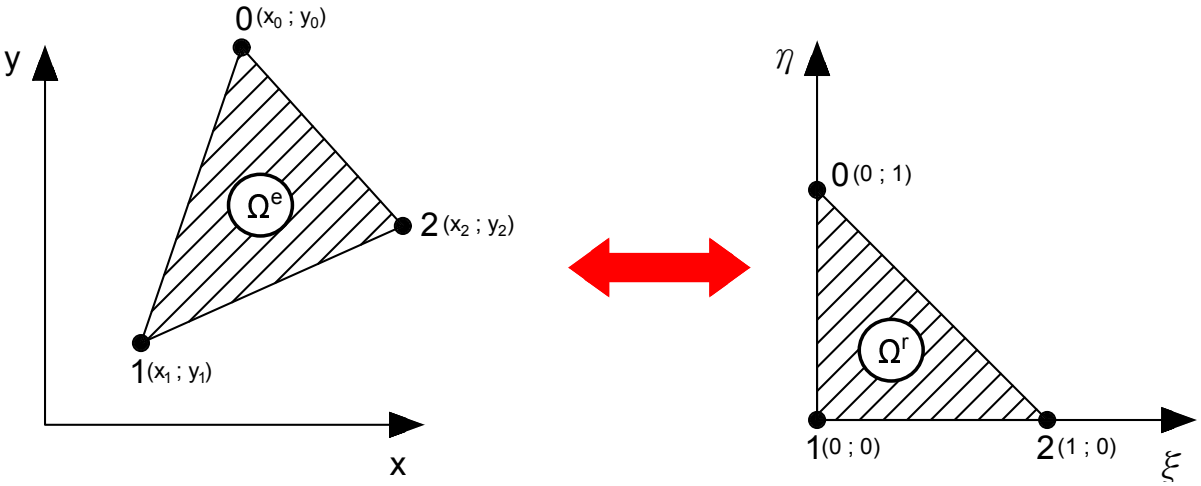
Dle typu rovinné úlohy je nutno dopočítat:

- **Rovinná napjatost** - složku tenzoru poměrné deformace $\boldsymbol{\varepsilon_z}$ užitím vztahu 11.
- **Rovinná deformace** - složku tenzoru napětí $\boldsymbol{\sigma_z}$ využitím vztahu 16.

4.3 Referenční prvky

4.3.1 Geometrická transformace souřadnic bodů prvku

Idea využití přirozených souřadnic spočívá ve vyjádření souřadnic a posuvů jednotlivých uzlů elementu pomocí stejných tvarových funkcí. Tyto funkce jsou definovány v bezrozměrném (homogenním) prostoru, který je dán přirozenou soustavou souřadnic (ξ, η). Každý bod elementu je jednoznačně určen bezrozměrnými souřadnicemi tak, aby jejich absolutní hodnoty nepřesáhli hodnotu jedné. Element, který je vytvořen pomocí přirozených souřadnic, je označován jako referenční nebo také rodičovský element. Výhoda použití referenčních prvků je dána zjednodušením při vyčíslování integrálu matice tuhosti. Transformace mezi jednotlivými souřadnicovými systémy je patrná z obrázku 7, kde je znázorněna pro tří-uzlový trojúhelníkový element.



Obrázek 7: Transformace tří-uzlového trojúhelníkového elementu ze skutečného prostoru (vlevo) do bezrozměrného prostoru (vpravo).

Transformace musí nutně splňovat tato pravidla:

a) Každému bodu elementu Ω^r musí odpovídat bod elementu Ω^e.

b) Geometrický uzel referenčního elementu Ω^r musí odpovídat geometrickému uzlu skutečného elementu Ω^e.

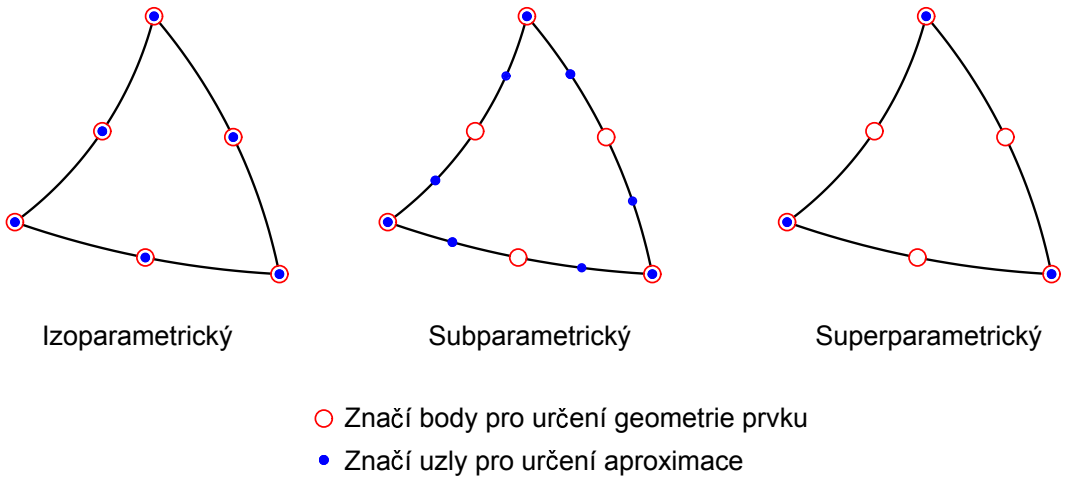
c) Hranice vymezená dvěma uzly referenčního elementu musí korespondovat s hranicí skutečného elementu vymezené odpovídajícími si uzly.

Matematicky lze transformaci geometrie vyjádřit rovnicemi

$$x(\xi,\eta) = \sum_{i=0}^{n} \overline{N}_i \, x_i, \qquad y(\xi,\eta) = \sum_{i=0}^{n} \overline{N}_i \, y_i. \tag{30}$$

Elementy mohou být na základě interpolačních funkcí rozděleny do tří základních skupin:

- **Izoparametrické** - geometrie elementu je transformována pomocí stejných tvarových funkcí jako primárně hledaný vektor posuvů $\hat{\boldsymbol{u}}$ apriori jsou identické geometrické a interpolační uzly.
- **Subparametrické** - geometrie elementu je transformována pomocí tvarových funkcí nižšího stupně polynomu.
- **Superparametrické** - geometrie elementu je transformována pomocí tvarových funkcí vyššího stupně polynomu.



Obrázek 8: Typy referenčních prvků.

Tato práce se však bude zabývat pouze prvky izoparametrickými, neboť jsou nejpoužívanější ve většině řešených případů. Bude tedy platit rovnost

$$\boldsymbol{N}(\xi,\eta) = \overline{\boldsymbol{N}}(\xi,\eta). \tag{31}$$

4.3.2 Tvorba tvarových funkcí

Funkci posunutí lze u referenčního elementu zapsat lineární kombinací na sobě nezávislých funkcí $F_0(\xi,\eta),\ F_1(\xi,\eta),...$ neboli

$$\hat{u}(\xi,\eta) = [F_0(\xi,\eta),\ F_1(\xi,\eta),\ ..\ ,F_n(\xi,\eta)]\begin{Bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{Bmatrix} = \boldsymbol{M}\ \boldsymbol{\alpha}. \tag{32}$$

Množina funkcí $F(\xi,\eta)$ vytváří takzvanou matici bázových funkcí $\boldsymbol{M}$ a vektor $\boldsymbol{\alpha}$ značí vektor obecných souřadnic.

Počet funkcí $F(\xi,\eta)$ se odvíjí od počtu stupňů volnosti elementu n. V tabulce 1 jsou uvedeny potřebné počty členů k sestavení polynomu r-tého stupně.

Stupeň polynomu r	Druh prvku		
	Jednorozměrný	Dvourozměrný	Třírozměrný
	n_d	n_d	n_d
1	2	3	4
2	3	6	10
3	4	10	20
4	5	15	35
5	6	21	56

Tabulka 1: Počet členů n_d potřebných pro sestavení polynomu r-tého stupně (7).

Z tabulky 1 je zřejmé, že kompletní polynomické funkce lze využít pouze v případě, kdy má element n_d stupňů volnosti. Pokud dojde k neshodě mezi počtem n a n_d mohou se vytvořit polynomy neúplné. Polynomické funkce je možné sestavit například s využitím *Pascalova trojúhelníku*.

Polynomy mohou být vytvořeny dvěma základními způsoby. První způsob sestavení aproximačního polynomu je spjat s existencí pouze obvodových uzlů. Tento typ prvků nese název *Serendepity family*.

Druhý způsob výběru polynomů je založen na tvorbě elementů, které obsahují i vnitřní uzly. Tyto elementy jsou označovány jako *Lagrange family*.

V tabulce 2 jsou uvedeny některé polynomické bázové funkce pro trojúhelníkový a čtyřúhelníkový element.

Trojúhelníkový element		
Stupeň polynomu	Polynomická bázová funkce	n_d
1	$[1, \xi, \eta]$ ... lineární	3
2	$[1, \xi, \eta, \xi^2, \eta^2, \xi\eta]$ ... kvadratický	6
Čtyřúhelníkový element		
Stupeň polynomu	Polynomická bázová funkce	n_d
2	$[1, \xi, \eta, \xi\eta]$ ... bilineární	4
3	$[1, \xi, \eta, \xi^2, \xi\eta, \eta^2, \xi^2\eta, \xi\eta^2]$ ... kvadratický	8

Tabulka 2: Polynomické bázové funkce pro trojúhelníkový a čtyřúhelníkový element.

Transformační funkce $N_(\xi, \eta)$ musí být stejné pro všechny souřadnice

$$\begin{aligned} x(\xi,\eta) &= [F_0(\xi,\eta), F_1(\xi,\eta), \ ..., \ F_n(\xi,\eta)]\boldsymbol{\alpha_x} = \boldsymbol{M}\,\boldsymbol{\alpha_x}, \\ y(\xi,\eta) &= [F_0(\xi,\eta), F_1(\xi,\eta), \ ..., \ F_n(\xi,\eta)]\boldsymbol{\alpha_y} = \boldsymbol{M}\,\boldsymbol{\alpha_y}. \end{aligned} \tag{33}$$

Jestliže je počet funkcí a koeficientů roven počtu stupňů volnosti elementu, transformace musí platit i pro uzlové body. Lze tedy psát

$$\boldsymbol{x_n} = \begin{Bmatrix} x_0 \\ x_1 \\ : \\ x_n \end{Bmatrix} = \begin{bmatrix} F_0(\xi,\eta) & F_1(\xi,\eta) & ... & F_n(\xi,\eta) \\ F_0(\xi,\eta) & F_1(\xi,\eta) & ... & F_n(\xi,\eta) \\ : & : & : & : \\ F_0(\xi,\eta) & F_1(\xi,\eta) & ... & F_n(\xi,\eta) \end{bmatrix} \begin{Bmatrix} \alpha_{0x} \\ \alpha_{1x} \\ : \\ \alpha_{nx} \end{Bmatrix} = \boldsymbol{A}\ \boldsymbol{\alpha_x}. \tag{34}$$

Vyjádřením vektoru koeficientů α_x z rovnice 34 a zpětným dosazením do rovnice 33 , potom bude

$$x(\xi,\eta) = \boldsymbol{M}\ \boldsymbol{A}^{-1}\ \boldsymbol{x_n} = \boldsymbol{N}(\xi,\eta)\,\boldsymbol{x_n}. \tag{35}$$

Analogicky lze psát

$$y(\xi,\eta) = \boldsymbol{N}(\xi,\eta)\,\boldsymbol{y_n}. \tag{36}$$

Aby bylo možné stanovit matici $\boldsymbol{B}(\xi,\eta)$ je třeba také provést transformaci deviátorové matice ∂ do referenčních souřadnic (ξ,η). S využitím věty o derivaci složené funkce a následnou inverzí vztahu vznikne rovnice

$$\begin{Bmatrix} \frac{f}{\partial x} \\ \frac{f}{\partial y} \end{Bmatrix} = \frac{1}{det(\boldsymbol{J})} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial y}{\partial \xi} \\ -\frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \xi} \end{bmatrix} \begin{Bmatrix} \frac{\partial f}{\partial \xi} \\ \frac{\partial f}{\partial \eta} \end{Bmatrix}, \tag{37}$$

kde $det(\boldsymbol{J})$ představuje determinant *Jakobiánu transformace*

$$det(\boldsymbol{J}) = \frac{\partial x}{\partial \xi}\frac{\partial y}{\partial \eta} - \frac{\partial x}{\partial \eta}\frac{\partial y}{\partial \xi}. \tag{38}$$

Vyčíslení determinantu Jakobiánu je možno provést dosazením rovnic 30 do rovnice 38

$$det(\boldsymbol{J}) = \sum_{i=0}^{n}\sum_{j=0}^{n} x_i \left(\frac{\partial N_i}{\partial \xi}\frac{\partial N_j}{\partial \eta} - \frac{\partial N_i}{\partial \eta}\frac{\partial N_j}{\partial \xi}\right) y_j. \tag{39}$$

Matice tuhosti elementu se využitím přirozených souřadnic stanoví jako

$$\boldsymbol{k} = \int_{-1}^{1}\int_{-1}^{1} \boldsymbol{B}^T \boldsymbol{D}\, \boldsymbol{B}\, t\, det(\boldsymbol{J})\, d\xi\, d\eta. \tag{40}$$

Pro výpočet matice tuhosti elementu je vhodné využít numerickou metodu, což spočívá v převedení integrálu na sumaci vhodným kvadraturním vzorcem. Nejpoužívanější metodou je *Gaussova metoda*, jejíž formální zápis je zřejmý z následující rovnice

$$\boldsymbol{k} = \sum_{i=1}^{m}\sum_{j=1}^{m} \Phi_i\, \Phi_j \boldsymbol{B}^T(\xi,\eta)\, \boldsymbol{D}\, \boldsymbol{B}^T(\xi,\eta)\, det(\boldsymbol{J})(\xi,\eta), \tag{41}$$

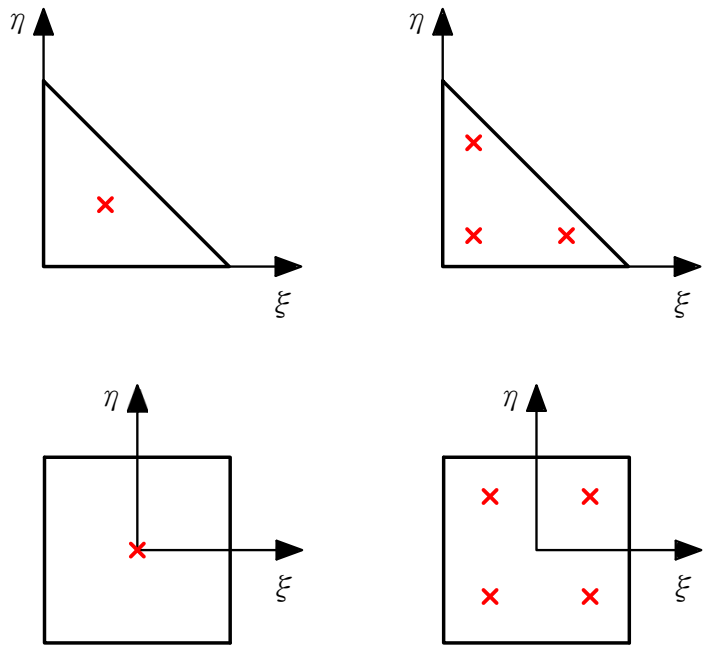
kde Φ jsou váhové koeficienty, které jsou určeny řádem kvadratury. Symbol m udává počet bodů, ve kterých je nutno vyčíslit integrand.

Hodnoty jednotlivých váhových koeficientů a souřadnic integračních bodů je možné najít v literatuře zabývající se numerickou matematikou či přímo numerickými metodami v mechanice (12). Pro ukázku byly do tabulky 3 zaneseny základní případy pro trojúhelníkový a čtyřúhelníkový element.

Trojúhelníkový element			
Řád integrace m	Souřadnice ξ_i	Souřadnice η_i	Váhové koef. Φ_i
1	0,33333 33333 333	0,33333 33333 333	0,50000 00000 000
3	0,16666 66666 667	0,16666 66666 667	0,33333 33333 333
	0,66666 66666 667	0,16666 66666 667	0,33333 33333 333
	0,16666 66666 667	0,66666 66666 667	0,33333 33333 333
Čtyřúhelníkový element			
Řád integrace m	Souřadnice ξ_i	Souřadnice η_i	Váhové koef. Φ_i
1	0,00000 00000 000	0,00000 00000 000	4,00000 00000 000
4	0,57735 02691 896	0,57735 02691 896	1,00000 00000 000
	−0,57735 02691 896	0,57735 02691 896	1,00000 00000 000
	−0,57735 02691 896	−0,57735 02691 896	1,00000 00000 000
	0,57735 02691 896	−0,57735 02691 896	1,00000 00000 000

Tabulka 3: Příklady parametrů pro Gaussovu integraci trojúhelníkového a čtyřúhelníkového elementu.

Pozici integračních bodů uvedených v tabulce 3 je možné pozorovat na obrázku 9. Počet zvolených integračních bodů ovlivňuje přesnost integrace.



Obrázek 9: Ukázka pozic integračních bodů pro trojúhelníkový a čtyřúhelníkový element.

5 Adaptivní techniky v MKP

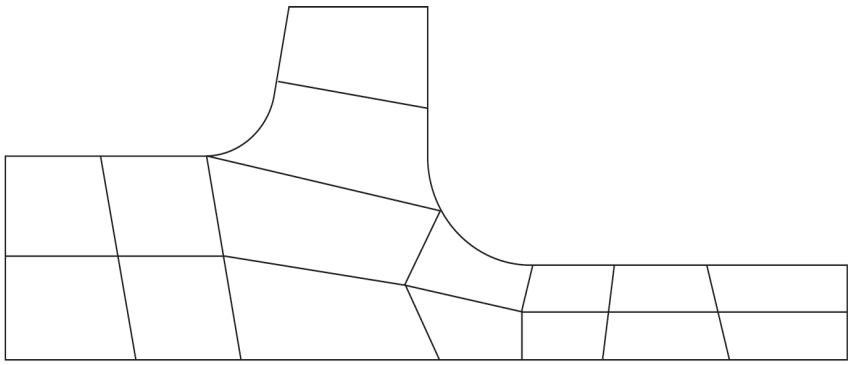
Adaptivní techniky v MKP jsou určeny k získání přesnějšího řešení vhodnou optimalizací konečno-prvkového modelu. Při řešení úlohy MKP je model diskretizován určitým počtem elementů různých typů. Jestliže výpočetní model nevykazuje dostatečnou přesnost, lze využít dvou základních metod zpřesnění výsledků.

První metodou je tvorba přesnějšího výpočtového modelu užitím více elementů stejného typu. Zvětšováním počtu prvků je zmenšován jejich charakteristický rozměr h. Pokud jsou splněny potřebné podmínky konvergence, výsledky poté konvergují k přesnému řešení. Tato adaptivní metoda je označována jako *h-verze* a dále je podle strategie „dělení“ rozčleněna do tří podskupin:

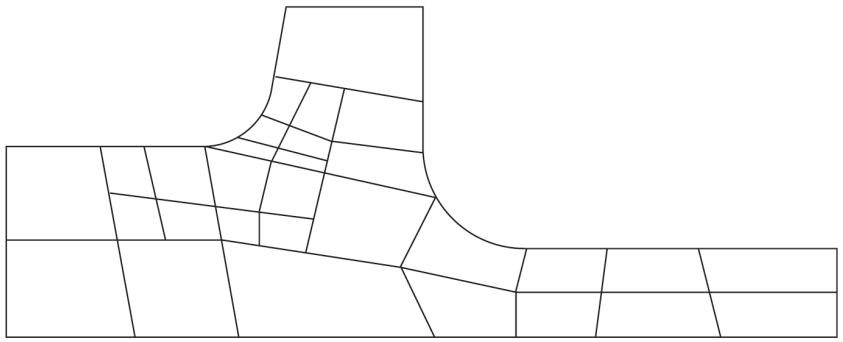
1. Široce využívaná strategie *h-verze* je v angličtině označována jako *enrichment*. Metoda spočívá v dělení elementů původní sítě, které nesplňují požadovanou přesnost, na elementy s menším charakteristickým rozměrem, aniž by došlo ke změně původní hranice elementu, viz obrázek 10(b).

2. Další strategií adaptace sítě založené na principu *h-verze* je kompletní přesíťování řešené oblasti, jež je v anglickém jazyce označováno jako *mesh regeneration* nebo *remeshing*. Metoda je založena na predikci optimální velikosti charakteristického rozměru elementů celé řešené oblasti z výsledků získaných na původní síti a následném vytvoření zcela nové sítě, jak je patrné na obrázku 10(c).

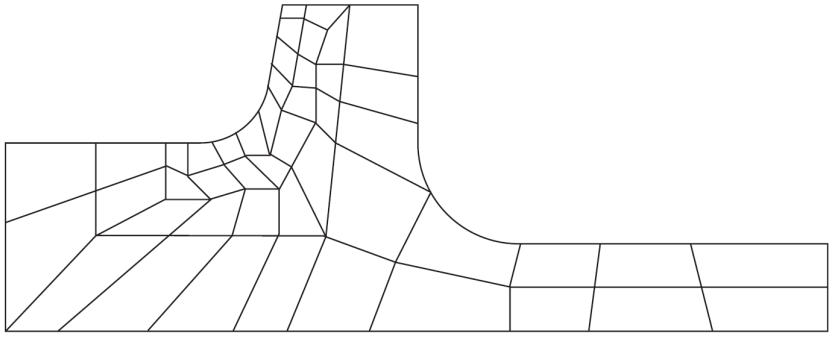
3. Poslední strategie, známá pod názvem *r-verze*, spočívá ve správném nastavení polohy původních uzlů (elementů), aniž by došlo ke změně jejich počtu, jak je zřejmé z obrázku 10(d). Ve skutečnosti se ale nejedná o pravou adaptivní strategii, neboť nemusí být dosaženo požadované přesnosti řešení (může dojít pouze k částečnému zpřesnění řešení).



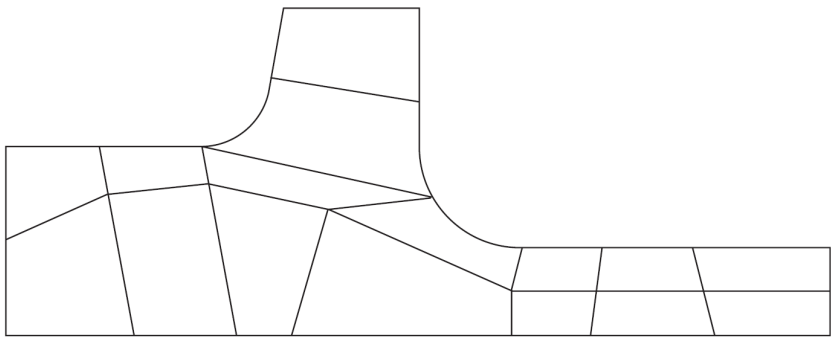
(a) Základní síť



(b) Dělení původních elementů (enrichment)



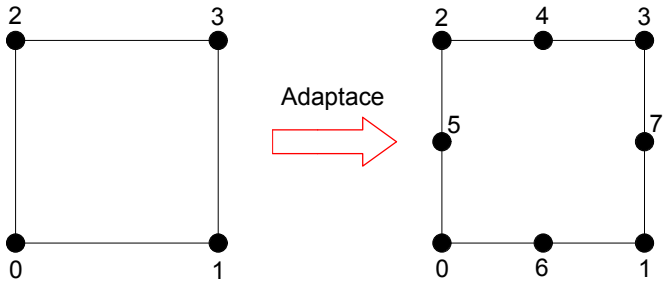
(c) Vytvoření zcela nové sítě (remeshing)



(d) Přeskupení uzlů původní sítě (r-verze)

Obrázek 10: Jednotlivé strategie adaptace sítě pomocí *h-verze* (13).

Druhým způsobem k dosažení přesnějších výsledků je *p-verze*. Tato metoda je založena na principu zvyšování stupně polynomické aproximace *p* elementu při zachování původní diskretizace modelu. Tím je zvyšován počet neznámých parametrů na elementu, a tím i řád příslušné matice tuhosti.



Obrázek 11: Příklad zvyšování stupně polynomické aproximace *p* na čtyřúhelníkovém elementu.

Na základě pokroku v teorii aposteriorních odhadů chyby řešení MKP bylo možno sestavit adaptivní algoritmy, jejichž postup je popsán dále.

Nejprve je proveden výpočet na vytvořené základní síti, na jehož výsledcích je zhotoven odhad chyby diskretizace řešení na každém elementu. U elementů, na nichž chyba přesahuje předem definovanou dovolenou hodnotu, je zmenšen charakteristický rozměr elementů *h* při užití *h-verze* nebo je zvýšen stupeň aproximačních funkcí *p* při využití *p-verze*. Poté je proveden nový výpočet s modifikovanou sítí. Postup je opakován, dokud u všech elementů neklesne chyba pod předepsanou hodnotu nebo do té doby, než je uživatelem dosaženo stanoveného maximálního počtu adaptací sítě.

Kvalitu adaptivního algoritmu je možno posoudit například dle rychlosti konvergence k požadované přesnosti řešení. Tento faktor bývá kontrolován počtem nutných iterací k dosažení požadované přesnosti řešení. Při využití *h-verze* je také sledován finální počet elementů, které bylo nutno vytvořit k vyřešení úlohy se stanovenou přesností. U *p-verze* by to byl naopak stupeň polynomické aproximace.

5.1 Adaptivní techniky dle Zienkiewicze a Zhua

Nejrozšířenějším způsobem pro identifikaci elementů vhodných k přesíťování je metoda dle Zienkiewicze a Zhua. Odhad chyby založené na jejich metodě je převážně určen pro *h – verzi* adaptivní techniky. Důvodem je možnost stanovení optimální velikosti charakteristického rozměru elementů, díky čemuž je zredukován počet nutných přesíťování k dosažení požadované přesnosti řešení.

5.1.1 Normy chyb dle Zienkiewicze a Zhua

Postup normování chyb včetně všech uvedených vztahů byl převzat z (13). Přibližná řešení posuvů $\hat{\boldsymbol{u}}$ a napětí $\hat{\boldsymbol{\sigma}}$ získaná pomocí metody konečných prvků se liší od exaktních hodnot $\boldsymbol{u}, \boldsymbol{\sigma}$. Rozdíl mezi těmito veličinami je chyba řešení. Pro posuvy je chyba stanovena rovnicí

$$\boldsymbol{e_u} = \boldsymbol{u} - \hat{\boldsymbol{u}}. \tag{42}$$

Pro poměrnou deformaci a napětí je chyba řešení dána vztahy

$$\boldsymbol{e_\varepsilon} = \boldsymbol{\varepsilon} - \hat{\boldsymbol{\varepsilon}}, \qquad \boldsymbol{e_\sigma} = \boldsymbol{\sigma} - \hat{\boldsymbol{\sigma}}. \tag{43}$$

Jde o definici chyby v bodě. Přednost je však dávána integrálním měřítkům chyb. V inženýrské praxi je využíváno integrální měřítko energetické normy. Energetickou normu chyby řešení lze vyjádřit jako:

$$\|e\| = \sqrt{\int_\Omega \boldsymbol{e_\varepsilon}^T \boldsymbol{D} \cdot \boldsymbol{e_\varepsilon} d\Omega} = \sqrt{\int_\Omega \boldsymbol{e_\varepsilon}^T \boldsymbol{e_\sigma} d\Omega} = \sqrt{\int_\Omega \boldsymbol{e_\sigma}^T \boldsymbol{D}^{-1} \boldsymbol{e_\sigma} d\Omega}. \tag{44}$$

Jednodušším integrálním měřítkem je L_2 norma, která může být použita pro jakoukoli veličinu. Pro posuv u je L_2 norma určena vzorcem

$$\|e_u\|_{L_2} = \sqrt{\int_\Omega \boldsymbol{e_u}^T \boldsymbol{e_u} d\Omega} \tag{45}$$

a pro napětí

$$\|e_\sigma\|_{L_2} = \sqrt{\int_\Omega \boldsymbol{e_\sigma}^T \boldsymbol{e_\sigma} d\Omega}. \tag{46}$$

Použitím L_2 normy je umožněn také výpočet střední kvadratické („root mean squeare“) chyby. Střední kvadratická hodnota chyby $\triangle u$ pro posuv na oblasti Ω je dána vztahem

$$|\triangle u| = \sqrt{\frac{\|e\|_{L_2}^2}{\Omega}}. \tag{47}$$

Podobně je vyjádřena střední kvadratická chyba pro napětí $\triangle\sigma$ na oblasti Ω

$$|\triangle\sigma| = \sqrt{\frac{\|e_\sigma\|^2_{L_2}}{\Omega}}. \tag{48}$$

Všechny výše zmíněné normy bývají vztaženy na celou řešenou oblast, podoblast či individuální element a to vztahem

$$\|e\|^2 = \sum_{i=1}^{m} \|e\|_i^2. \tag{49}$$

Pro lepší interpretaci lze využít relativní procentuální chybu, která je dána jako

$$\psi = \frac{\|e\|}{\|U\|} \times 100\%, \tag{50}$$

kde

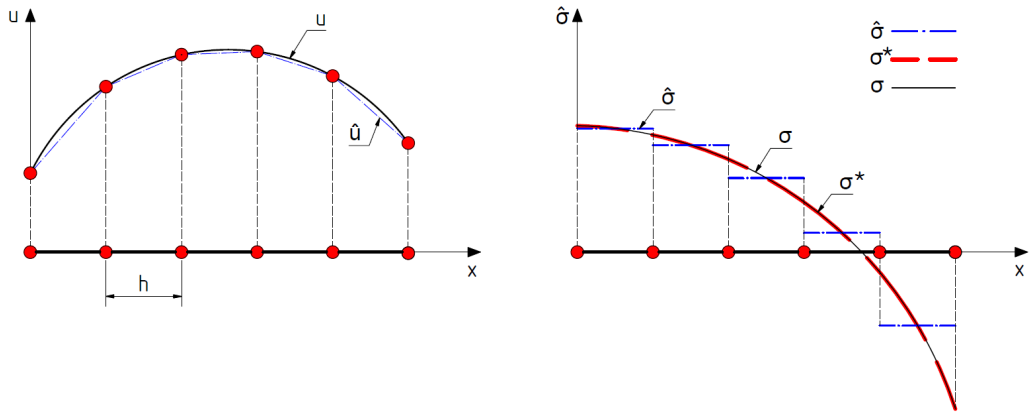
$$\|U\| = \sqrt{\int_\Omega \boldsymbol{\sigma}^T \cdot \boldsymbol{D}^{-1} \boldsymbol{\sigma} d\Omega} \tag{51}$$

je energetická norma řešení.

Vytvořený algoritmus využívá k odhadu chyby nad elementem rozdíl vypočteného napětí nad elementem a *vyhlazeného napětí* v uzlech elementu. Postup výpočtu vyhlazeného napětí je popsán v následující kapitole.

5.1.2 Vyhlazení napětí nad elementy

Aproximace funkcí posuvu založených na principu lineárních prvků má za následek nespojitost v napětích $\hat{\sigma}$. Na obrázku 12 je znázorněna situace při řešení jednorozměrné úlohy (tyče).



Obrázek 12: Průběh posuvu a napětí při řešení jednorozměrné úlohy.

Je zde patrné, že posuv $\hat{\boldsymbol{u}}$ se mění po prvku lineárně, kdežto napětí $\hat{\boldsymbol{\sigma}}$ je po prvku konstantní (není spojité). Proto byl Zienkiewiczem a Zhuem zaveden předpoklad (intuitivně), že napětí $\boldsymbol{\sigma}^*$ je interpolováno na prvku stejným způsobem jako posuvy $\hat{\boldsymbol{u}}$. Tato intuice byla později matematicky potvrzena. Na izolovaném prvku poté platí

$$\boldsymbol{\sigma}^* = \boldsymbol{N}\boldsymbol{r}_{\sigma}^{\boldsymbol{e}}, \tag{52}$$

kde $\boldsymbol{r}_{\sigma}^{\boldsymbol{e}}$ představuje vektor napětí v uzlových bodech, který je však neznámou. Pro jeho určení bylo využito vztahu pro střední kvadratickou chybu nad elementem

$$E_e = \frac{1}{2}\int_e (\boldsymbol{\sigma}^* - \hat{\boldsymbol{\sigma}})^2 \, dA. \tag{53}$$

Pokud je závorka ze vztahu 53 umocněna dle patřičného vzorce a je provedena separace jednotlivých částí integrálu, obdrží se vztah

$$E_e = \frac{1}{2}\int_e \boldsymbol{\sigma}^{*2} \, dA - \int_e \boldsymbol{\sigma}^* \hat{\boldsymbol{\sigma}} \, dA + \frac{1}{2}\int_e \hat{\boldsymbol{\sigma}}^2 \, dA. \tag{54}$$

Po dosazení 52 do 54 lze psát

$$E_e = \frac{1}{2}\boldsymbol{r}_{\sigma}^{\boldsymbol{e}\,T}\int_e \boldsymbol{N}^T\boldsymbol{N}\boldsymbol{r}_{\sigma}^{\boldsymbol{e}} \, dA - \boldsymbol{r}_{\sigma}^{\boldsymbol{e}\,T}\hat{\boldsymbol{\sigma}}\int_e \boldsymbol{N}^T \, dA + \frac{1}{2}\int_e \hat{\boldsymbol{\sigma}}^2 \, dA. \tag{55}$$

Následně je provedena minimalizace rovnice 55 dle vektoru neznámých napětí v uzlech elementu $\boldsymbol{r}_{\sigma}^{\boldsymbol{e}}$, přičemž poslední člen v rovnici 55 je konstanta, tudíž je jeho derivace nulová

$$\frac{\partial E_e}{\partial \boldsymbol{r}_{\sigma}^{\boldsymbol{e}}} = 0 = \int_e \boldsymbol{N}^T\boldsymbol{N}\boldsymbol{r}_{\sigma}^{\boldsymbol{e}} \, dA - \hat{\boldsymbol{\sigma}}\int_e \boldsymbol{N}^T \, dA. \tag{56}$$

Pro větší přehled byly zavedeny následující substituce

$$\boldsymbol{W}^{\boldsymbol{e}} = \int_e \boldsymbol{N}^T\boldsymbol{N} \, dA, \tag{57}$$

$$\boldsymbol{V}^{\boldsymbol{e}} = \hat{\boldsymbol{\sigma}}\int_e \boldsymbol{N}^T \, dA. \tag{58}$$

Nad jedním elementem poté platí

$$\boldsymbol{W}^{\boldsymbol{e}}\boldsymbol{r}_{\sigma}^{\boldsymbol{e}} = \boldsymbol{V}^{\boldsymbol{e}}. \tag{59}$$

Obdobně lze sestavit systém rovnic pro celou řešenou oblast

$$\boldsymbol{W}\boldsymbol{r}_{\sigma} = \boldsymbol{V}, \tag{60}$$

kde $\boldsymbol{r}_{\sigma} = (r_{\sigma_1}, r_{\sigma_2}, ..., r_{\sigma_N})^T$ je vektor neznámých napětí v uzlových bodech celé řešené oblasti. Tato soustava rovnic může být řešena stejnými způsoby, které byly použity pro výpočet posuvů v kapitole 6.4.

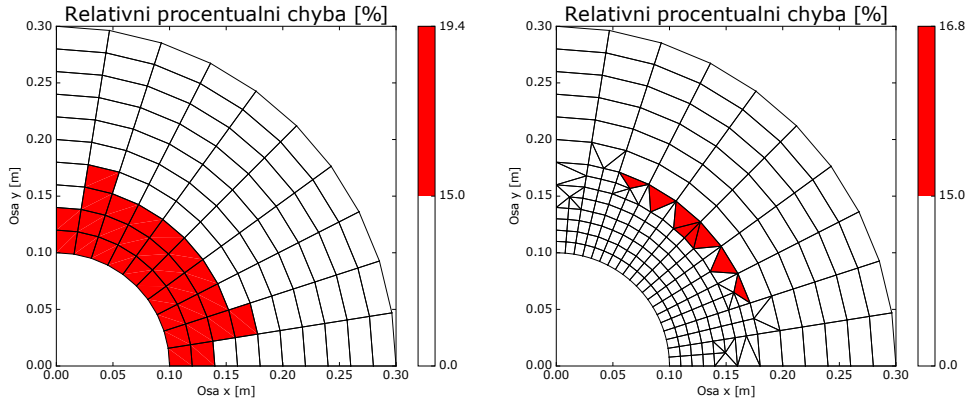
5.1.3 Proces adaptace sítě

Strategie zjemňování sítě je založena na požadavku dosažení předepsané procentuální chyby energetické normy a to na celé oblasti řešení. Tedy aby platilo

$$\psi \leq \bar{\psi}, \tag{61}$$

kde $\bar{\psi}$ je maximální přípustná procentuální chyba.

Tato strategie je zachycena na obrázku 13, jež znázorňuje řešení tlustostěnné nádoby s maximální přípustnou procentuální chybou $\bar{\psi} = 15\%$. V levé části obrázku 13 je znázorněno rozložení vypočtené relativní procentuální chyby, kde jsou červenou barvou zvýrazněny elementy, které překračují hodnotu přípustné chyby. Tyto elementy jsou v následujícím kroku adaptovány, což má za následek pokles maximální hodnoty relativní chyby. Tento proces by dále pokračoval, dokud by na všech elementech neklesla hodnota relativní procentuální chyby pod stanovenou přípustnou hodnotu.



Obrázek 13: Stanovená relativní procentuální chyba na počáteční síti (vlevo), stanovená relativní procentuální chyba na adaptované síti (vpravo).

Jelikož jde o tlustostěnnou nádobu, dalo by se předpokládat axisymetrické rozložení relativní procentuální chyby a tudíž i axisymetricky adaptovaná síť. Jelikož se však jedná o metodu konečných prvků, kde je zatížení od vnějšího a vnitřního tlaku transformováno do uzlů, vznikají v uzlech, kde jsou aplikovány mimo jiné také okrajové podmínky symetrie, tlakové ztráty. To znamená, že dochází ke ztrátě jedné zatěžující složky (ve směru osy x či y).

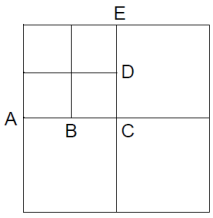
5.2 Adaptivní techniky v komerčních MKP programech

Součástí diplomové práce byla mimo jiné také komparace efektivity vytvořeného MKP algoritmu s komerčními MKP programy. Pro tento účel byly zvoleny dva velmi populární a hojně používané komerční MKP programy s podporou adaptivních technik.

5.2.1 Marc Mentat

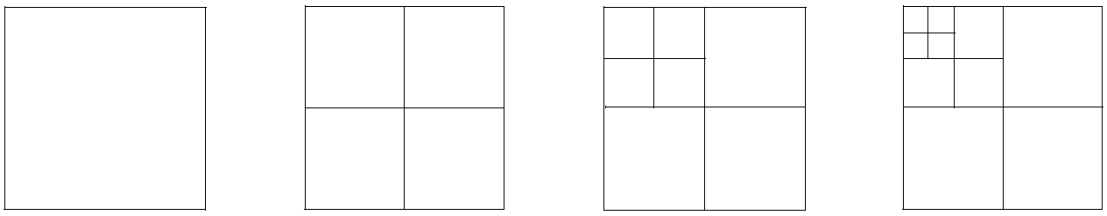
Prvním programem, který byl zvolen, je Marc Mentat. Tímto programem je nabízeno použití adaptivních technik jak v případě lineární tak i nelineární úlohy. Pro využití této funkce je nutno mít síť tvořenou pouze elementy nižšího řádu, což jsou v případě 2-D úlohy 3-uzlové trojúhelníky či 4-uzlové čtyřúhelníky.

Samotná adaptace sítě probíhá dělením elementů a následným vnitřním svázáním uzlů pro zajištění kompatibility. Uzel B je efektivně svázán s uzly A, C a uzel D je efektivně svázán s uzly C, E. Tato svázání nemají žádný vliv na uživatelem definované kontakty.



Obrázek 14: Proces svázání uzlů při adaptaci sítě v programu Marc Mentat(15).

Na obrázku 15 je zachycen proces adaptace sítě pro čtyřúhelníkový element.



Obrázek 15: Proces zjemnění sítě v programu Marc Mentat (15).

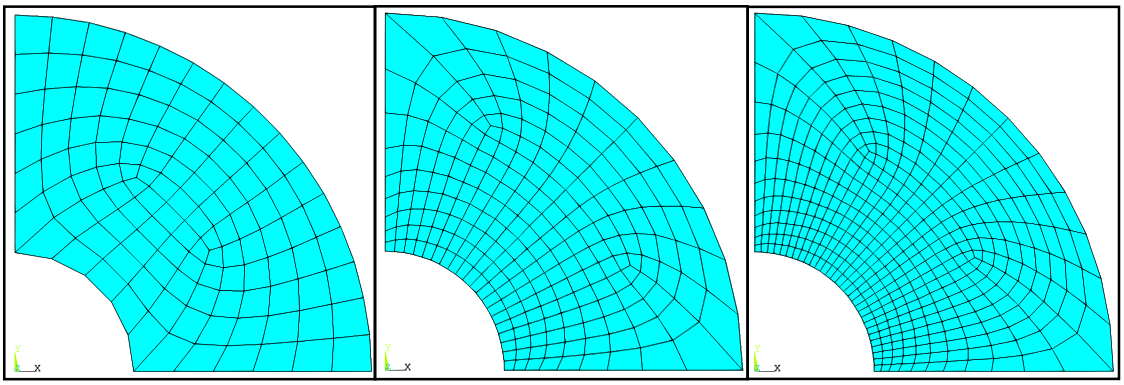
Pro lokální adaptaci lze v programu Marc Mentat zvolit hned několik kritérií, dle kterých se budou vybírat elementy vhodné k adaptaci. Pro část validace byl zvolen aposteriorní odhad chyby napětí s využitím metody Zienkiewitcze a Zhua.

5.2.2 ANSYS APDL

Druhou volbou byl ANSYS APDL. Tento program byl zvolen na základě možnosti vytvoření základní sítě z tří-uzlových či čtyř-uzlových elementů a následnou specifikaci adaptace sítě. Ta je v programu ANSYS APDL také založena na aposteriorním odhadu chyby napětí s využitím metody Zienkiewicze a Zhua. Samotná adaptace sítě je v programu spuštěna užitím makra *ADAPT*.
Aby mohlo dojít k adaptaci, musí být modelem splněna určitá kritéria:

- Adaptace lze provést pouze v případě lineárně statické analýzy.
- Model musí být definován pouze jedním typem materiálu, neboť stanovení chyby je založeno na výpočtu vyhlazeného napětí (Average nodal stress), které by na materiálovém rozhraní nebylo správně vypočteno.

Program ANSYS APDL má zcela odlišný způsob adaptace sítě než předchozí program Marc Mentat. Po stanovení hodnoty relativní procentuální chyby nad elementem je vyhodnocena optimální délka hrany elementy a následně je vytvořena zcela nová síť. Z tohoto důvodu nemůže být počáteční síť vytvořena užitím přímého dělní hran na určitý počet elementů. Proces adaptace sítě v programu ANSYS APDL je zachycen na obrázku 16, kde je předveden na úloze tlustostěnné nádoby.



Obrázek 16: Proces adaptace sítě v programu ANSYS APDL - počáteční síť (vlevo), první adaptace (střed), druhá adaptace (vpravo).

6 Implementace vlastního konečno-prvkového algoritmu v prostředí programu Python

Implementace vlastního konečno-prvkového programu byla uskutečněna v programovacím jazyce *Python*. Jedná se o hybridní nebo také více-paradigmatický jazyk, což znamená, že uživateli umožňuje používat *objektově orientovaná*, *procedurální* i *funkcionální* paradigmata dle typu úlohy.

Velkou předností programovacího jazyka *Python* je jeho jednoduchost a zároveň jeho velká produktivita z hlediska psaní programů. Další velkou výhodou je dostupnost a velice jednoduchá použitelnost široké škály modulů, které uživateli usnadňují řešení úloh z celé řady oblastí. Pro implementaci MKP algoritmu byly využity převážně knihovny s názvem *numPy*, *SciPy* a *Matplotlib*.

NumPy je základní balíček určený převážně pro vědecké výpočty. Obsahuje nástroje jako:

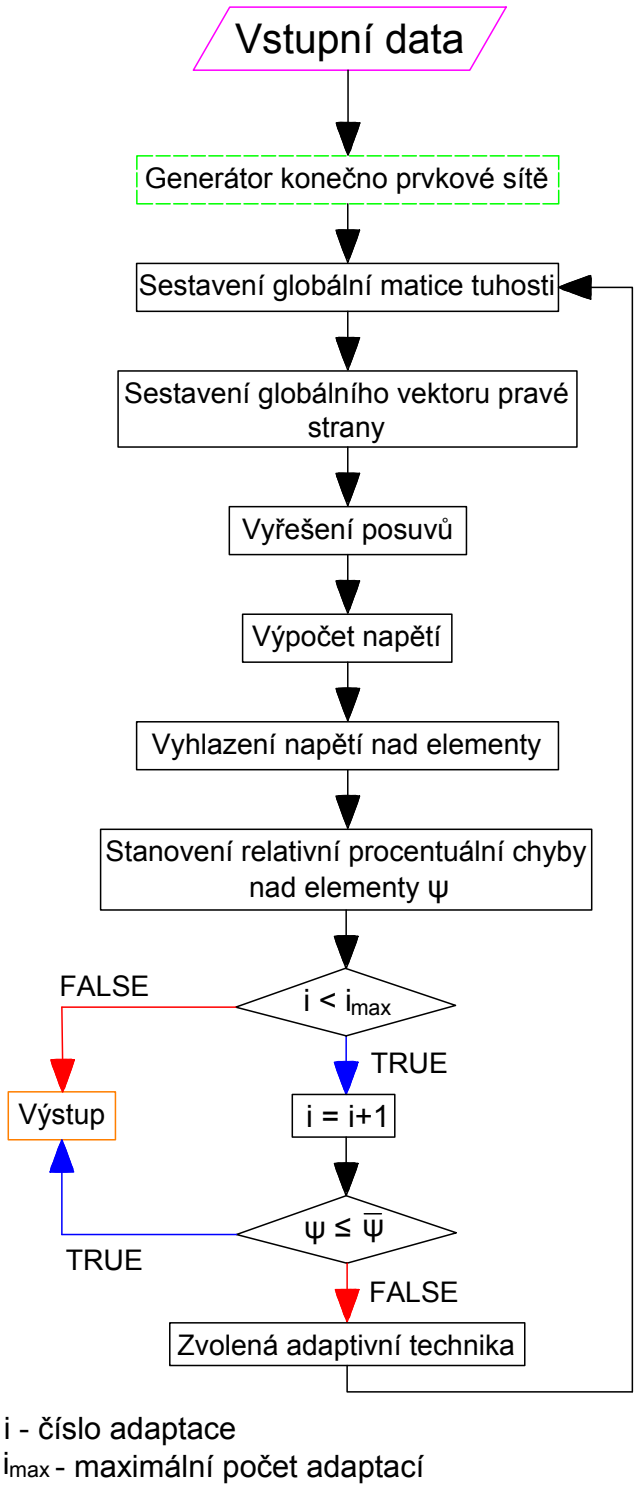
- vytváření N-dimenzionálních polí,
- sofistikované vytváření funkcí,
- nástroje pro interakci s *C/C++* či *Fortran* programovacími jazyky.

SciPy je jedna z hlavních knihoven pro vědecké výpočty. Poskytuje uživatelům jednoduché a efektivní prostředky pro numerickou integraci či optimalizaci. Další významnou součástí knihovny *SciPy* je manipulace s řídkými maticemi, jež jsou velmi významnou součástí implementovaného algoritmu.

Matplotlib je knihovna obsahující grafické nástroje pro tvorbu velkého množství grafických výstupů. Pomocí této knihovny lze vytvářet například histogramy, sloupcové grafy nebo chybové grafy a to pouze na pár řádcích kódu. Pomocí této knihovny jsou vytvářeny všechny grafické výstupy z implementovaného MKP algoritmu.

6.1 Vlastní MKP algoritmus

Vytvořený MKP algoritmus je určen k výpočtům dvoudimenzionálních úloh se sítí tvořenou tří-uzlovými či čtyř-uzlovými elementy nebo jejich kombinací. Princip algoritmu je zachycen na obrázku 17.



Obrázek 17: Schéma vlastního MKP algoritmu.

Nejprve je tedy nutno nastavit všechny potřebné informace vstupující do výpočtu jako jsou:

- **Materiálové parametry** - Modul pružnosti v tahu E, Poissonovo číslo μ.
- **Stav rovinné úlohy** - Rovinná napjatost či rovinná deformace.
- **Informace o konečno-prvkové síti** - V případě tlustostěnné nádoby je zde zadáván počet elementů po obvodě n_o a tloušťce nádoby n_t, dále její vnitřní průměr R_0 a vnější průměr R_1.
- **Aplikace okrajových podmínek** - V případě tlustostěnné nádoby se zadává hodnota tlaku na vnitřní stěnu P_0 a na vnější stěnu P_1.
- **Volba algoritmu pro adaptaci sítě** - Lze zvolit ze dvou algoritmů pracujících na h-verzi adaptace sítě. První algoritmus s názvem *Standardní strategie dělení elementů* je vhodný pro oba typy elementů (včetně jejich kombinace). Druhý algoritmus nesoucí jméno *Modifikovaná strategie dělení elementů* je určený pouze pro síť tvořenou čtyřúhelníkovými elementy. Jejich specifikace je rozebrána dále v textu.
- **Nastavení parametrů adaptace sítě** - Nutno zvolit maximální přípustnou procentuální chybu na elementu $\bar{\psi}$ a maximální počet adaptací sítě.
- **Výběr řešiče** - Zde je možno vybrat metodu pro řešení systému rovnic. Na výběr je explicitní metoda *LU transformace* a to s využitím plné nebo řídké matice tuhosti a dále také numerická metoda *sdružených gradientů*, která využívá pouze matice v řídkém formátu.
- **Grafický výstup** - Na závěr je nutno zvolit grafický výstup. Jsou zde předdefinované algoritmy pro vykreslení všech vyskytujících se napětí, celkové i směrové posunutí a také vypočtená chyba na elementech.

Po definici vstupních dat již začíná algoritmus pracovat. V případě tlustostěnné nádoby jsou nejprve vygenerována pole nesoucí informace o elementech a pozicích jednotlivých uzlů. Pro jiné úlohy je nutno tato pole naimportovat ve vhodném formátu.

Tato pole jsou určena jako vstup pro sestavování globální matice tuhosti $\boldsymbol{K}$, která je vždy sestavována v řídkém formátu *CSC*. Pokud je zvolen řešič, jež využívá plnou matici tuhosti, matice tuhosti je pouze převedena do plného formátu.

Poté následuje aplikace okrajových podmínek spolu se sestavením globálního vektoru pravé strany $\boldsymbol{F}$, čímž jsou připraveny všechny vstupy pro vyřešení neznámých uzlových posuvů.

Po jejich vyřešení je možno dopočítat sekundární neznámé jako jsou poměrné deformace a napětí. Při výpočtu napětí na čtyř-uzlových elementech jsou napětí ještě extrapolována z integračních bodů do uzlů. U tří-uzlových trojúhelníkových elementů tato extrapolace není nutná, neboť k numerické integraci nedochází.

Jakmile jsou vypočtena napětí σ_x, σ_y a τ_{xy} pro všechny elementy, následuje proces vyhlazování napětí, tak aby byly spojité na celé řešené oblasti. Pokud je úloha typu rovinné deformace je zde také vypočteno třetí normálové napětí σ_z.

Na závěr je stanovena relativní chyba nad elementy, která je používána k identifikaci elementů vhodných k adaptaci. Relativní chyba je stanovena na základě teorie Zienkiewicze a Zhua, jež byla uvedena v kapitole 5.

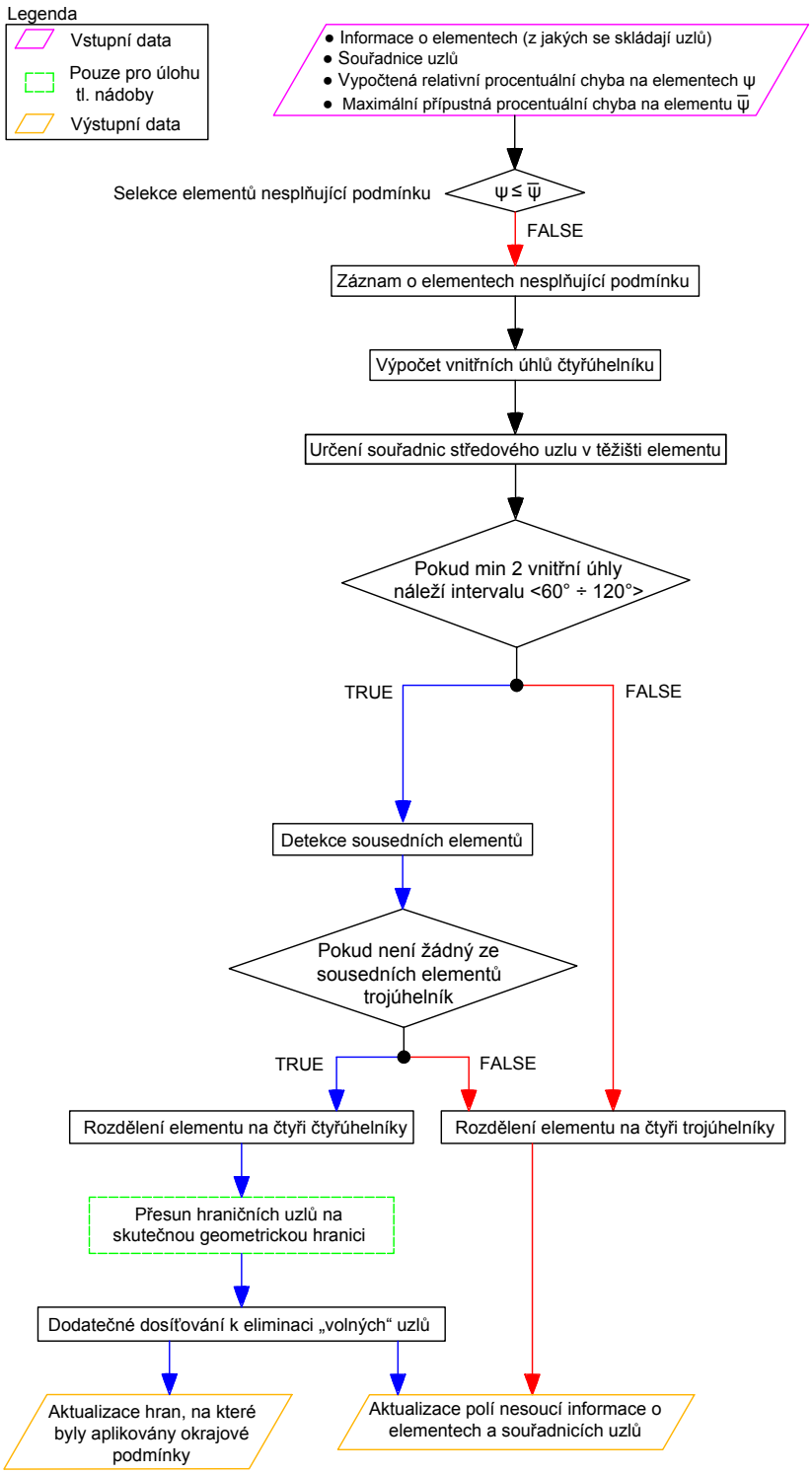
Pokud na některém z elementů vypočtená relativní procentuální chyba přesahuje přípustnou procentuální chybu a zároveň není dosaženo maximálního počtu adaptací, jsou všechny takto vytypované elementy adaptovány zvoleným algoritmem. Proces je opakován dokud na všech elementech není dosaženo požadované přesnosti nebo dokud není dosaženo maximálního počtu adaptací.

6.2 Standardní strategie dělení původních elementů

Algoritmus slouží pro adaptaci trojúhelníkové nebo čtyřúhelníkové sítě, ale také pro jejich kombinaci. Nejprve je vždy provedena adaptace čtyřúhelníkových elementů načež následuje adaptace trojúhelníkových elementů. Pro větší přehlednost zde byly algoritmy uvedeny zvlášť, ačkoli v implementovaném MKP algoritmu pracují v kooperaci.

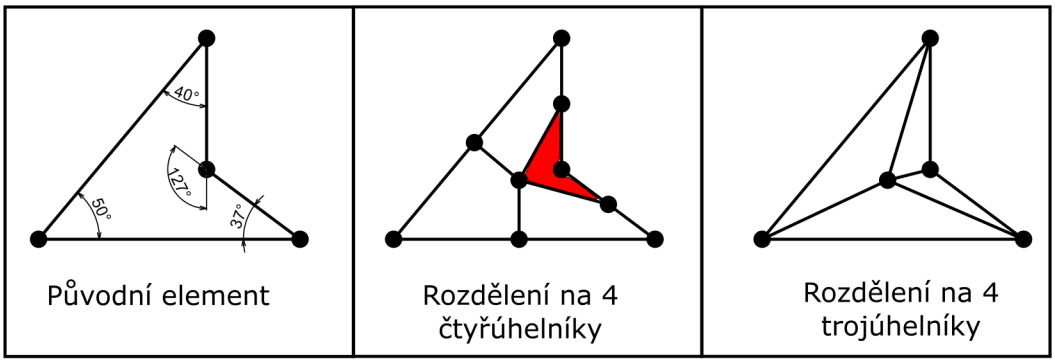
6.2.1 Dělení čtyřúhelníkových elementů

Tato část algoritmu spočívá v dělení vytypovaných čtyřúhelníkových elementů na menší elementy a následném dotváření sítě tak, aby nebyly přítomny žádné volné uzly. Schéma algoritmu je patrno z obrázku 18, přičemž samotný algoritmus je uveden v příloze 8.



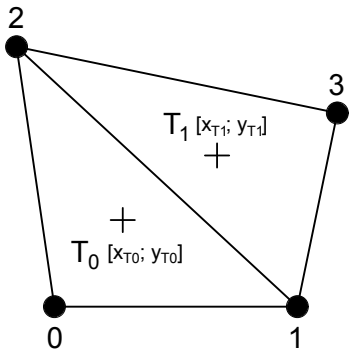
Obrázek 18: Standardní strategie dělení čtyřúhelníkových elementů.

Elementy, které jsou určeny k přesíťování (přesáhli maximální hodnotu přípustné chyby $\bar{\psi}$), jsou nejprve podrobeny kontrole na velikost všech vnitřních úhlů. Pokud více jak dva vnitřní úhly čtyřúhelníku leží mimo interval $\langle 60°; 120° \rangle$ není vhodné provádět dělení na čtyři menší čtyřúhelníky, neboť by mohly být značně zdeformovány. Proto je vhodnější takový čtyřúhelník rozdělit na čtyři trojúhelníky, čímž je předcházeno generaci zdeformovaných elementů. Tento stav je zachycen na obrázku 19, kde je červenou barvou znázorněn tvarově nevhodný element po provedení adaptace původního elementu.



Obrázek 19: Kontrola vnitřních úhlů čtyřúhelníku.

Po stanovení vnitřních úhlů následuje vlastní proces adaptace elementu, kdy je nejprve vypočtena pozice středového uzlu, jež je vždy umístěn do těžiště elementu. Pro jeho určení se element pomyslně rozdělí na dva trojúhelníky, viz obrázek 20.



Obrázek 20: Pomyslné rozdělení čtyřúhelníku na dva trojúhelníky za účelem určení pozice těžiště.

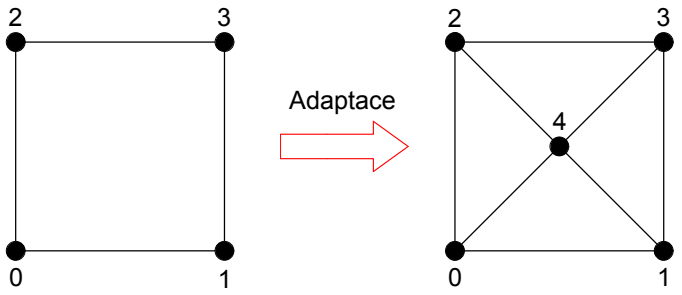
Souřadnice těžiště trojúhelníků se stanoví z rovnice 62, kde i značí číslo trojúhelníku

$$x_{Ti} = \frac{x_i + x_{i+1} + x_{i+2}}{3}, \qquad y_{Ti} = \frac{y_i + y_{i+1} + y_{i+2}}{3}. \tag{62}$$

Samotné souřadnice těžiště čtyřúhelníku jsou následně stanoveny pomocí váženého průměru jednotlivých těžišť trojúhelníků

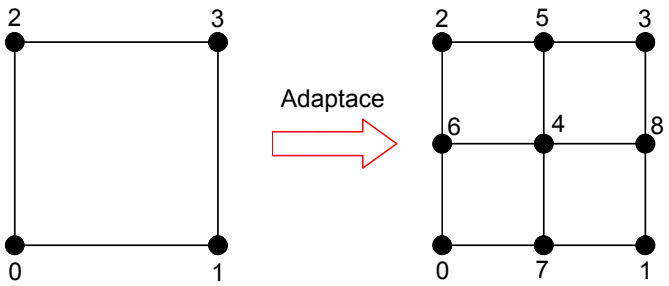
$$x_T = \frac{\sum_{i=0}^{1} x_{Ti}\, S_i}{\sum_{i=0}^{1} S_i}, \qquad y_T = \frac{\sum_{i=0}^{1} y_{Ti}\, S_i}{\sum_{i=0}^{1} S_i}, \tag{63}$$

Pokud vnitřní úhly nesplnily požadovanou podmínku nebo pokud je alespoň jeden sousední element trojúhelník, je element rozdělen na čtyři trojúhelníky. Po adaptaci je vždy provedena aktualizace polí nesoucích informace o elementech a uzlech. Nevýhodou tohoto dělení je poměrně slabá redukce relativní procentuální chyby elementu ψ, což může navýšit počet iterací nutných k dosažení požadované přesnosti.



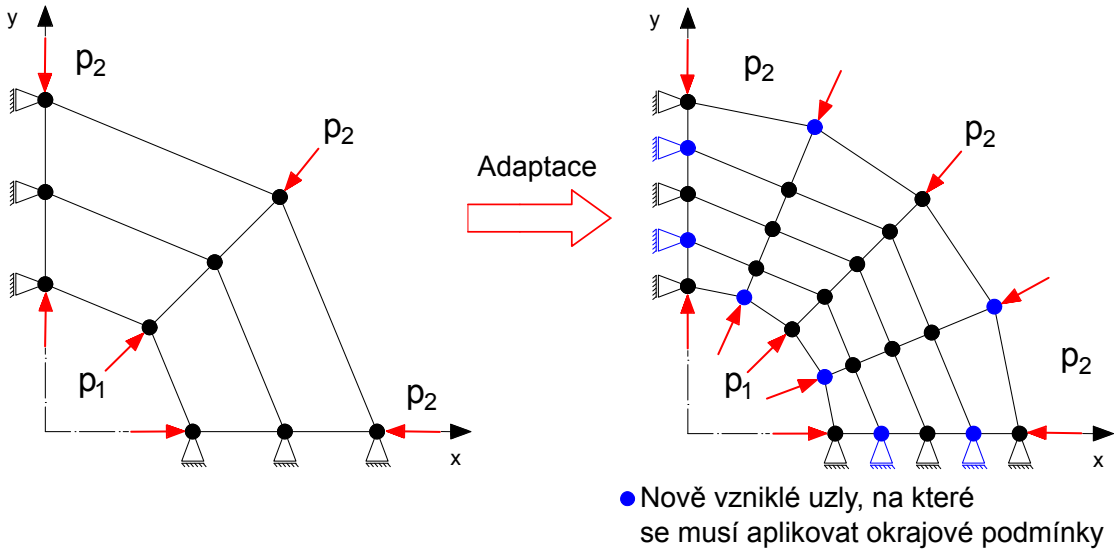
Obrázek 21: Dělení čtyřúhelníkového elementu na 4 trojúhelníky.

Jestliže alespoň dva vnitřní úhly elementu náleží stanovenému intervalu a všechny sousední elementy jsou čtyřúhelníky je element vhodný k rozdělení na čtyři menší čtyřúhelníky.



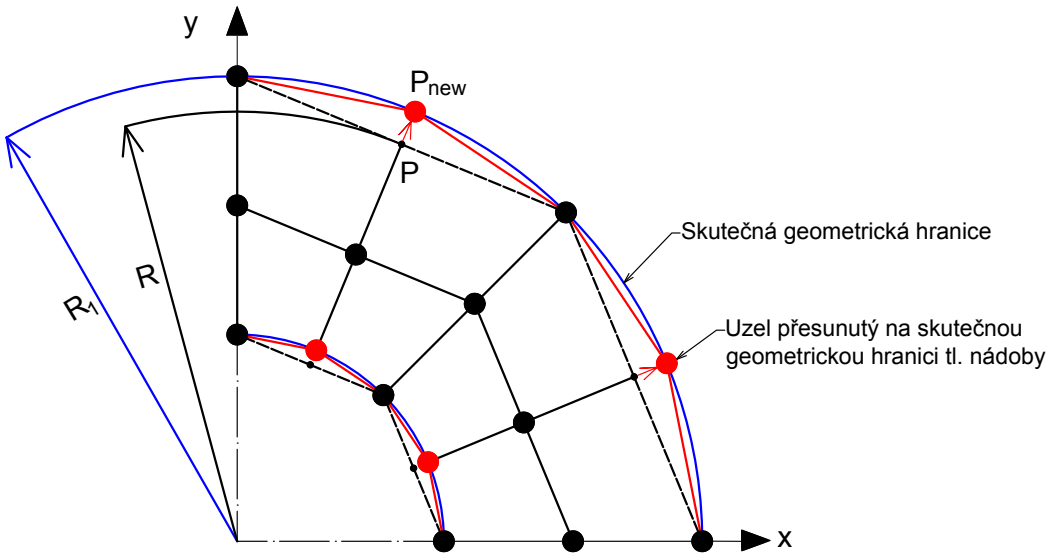
Obrázek 22: Dělení čtyřúhelníkového elementu na 4 čtyřúhelníky.

Protože jsou tímto algoritmem tvořeny nové uzly ve středech hran původního elementu, je nutno kromě polí nesoucích informace o elementech a uzlech aktualizovat i pole s informacemi o hranách, na nichž jsou aplikovány okrajové podmínky.



Obrázek 23: Nově vzniklé uzly náležící hranám, na které jsou aplikovány okrajové podmínky.

Pro úlohu tl. nádoby byl taktéž zakomponován algoritmus pro přesun nově vzniklých uzlů ve středech hran na skutečnou geometrickou hranici, viz obr. 24.



Obrázek 24: Přesun nově vzniklých uzlů ve středech hran na skutečnou geometrickou hranici.

Například, pro vnější poloměr tlustostěnné nádoby R_1 lze psát rovnici

$$R_1 = ||\boldsymbol{P}_{new}|| = ||\boldsymbol{P}||\, c, \tag{64}$$

kde $\boldsymbol{P} = \{x,\ y\}^T$ je vektor souřadnic uzlu před přesunem, $\boldsymbol{P}_{new} = \{x_{new},\ y_{new}\}^T$ je vektor souřadnic již přesunutého uzlu a c je konstanta, určující poměr jednotlivých vzdáleností od počátku. Jestliže se konstanta c vyjádří z rovnice 64

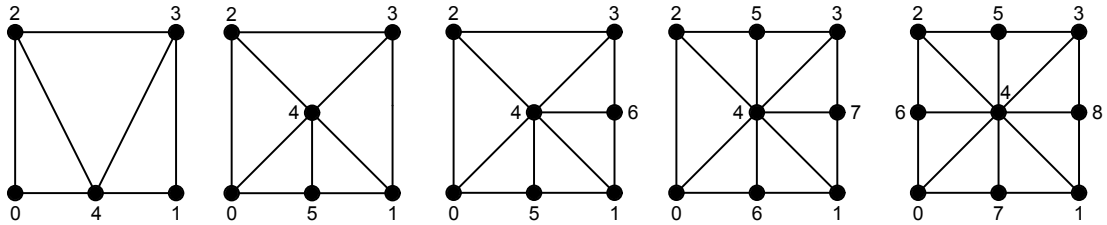
$$c = \frac{||\boldsymbol{P}_{new}||}{||\boldsymbol{P}||}, \tag{65}$$

je možno následně stanovit nový vektor souřadnic $\boldsymbol{P}_{new}$ dosazením vypočtené konstanty c a známého vektoru souřadnic $\boldsymbol{P}$ do rovnice

$$\boldsymbol{P}_{new} = \boldsymbol{P}\, c. \tag{66}$$

Tímto přesunem uzlů je umožněno řešiteli vytvořit poměrně hrubou počáteční síť, kterou je po adaptaci poskytnuto řešení s požadovanou přesností. Jelikož je tato funkce dostupná pouze pro řešení tlustostěnné nádoby, je nutno při řešení jiných úloh dbát na dostatečný popis geometrie vytvořenou konečno-prvkovou sítí.

Po provedení adaptace všech elementů k tomu určených, je potřeba provést dodatečné dosíťování čtyřúhelníkových elementů, kterými jsou eliminovány všechny volné uzly. K tomuto dosíťování čtyřúhelníkových elementů jsou využívány strategie dělení na trojúhelníky, jejichž počet je dán konkrétní situací. Všechny možnosti dodatečného dosíťování čtyřúhelníkových elementů jsou patrné z obrázku 25.

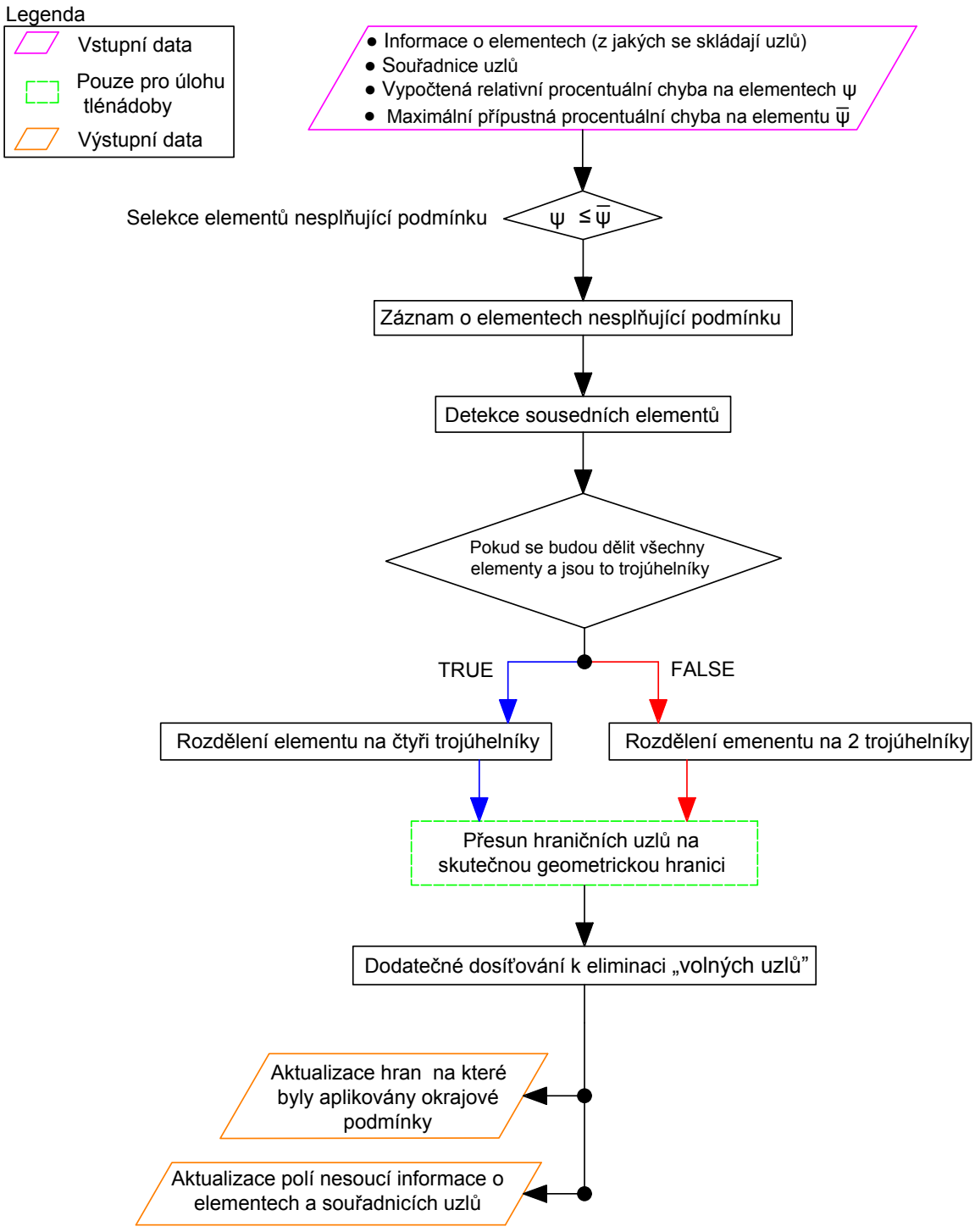


Obrázek 25: Implementované možnosti adaptace čtyřúhelníkových elementů při dosíťování k eliminaci volných uzlů.

První způsob dotváření sítě, jež je zachycen na obrázku 25, byl kvůli neefektivnosti snižování chyby v algoritmu potlačen. Je zde však možnost jeho odkomentování a zpětného zařazení do algoritmu.

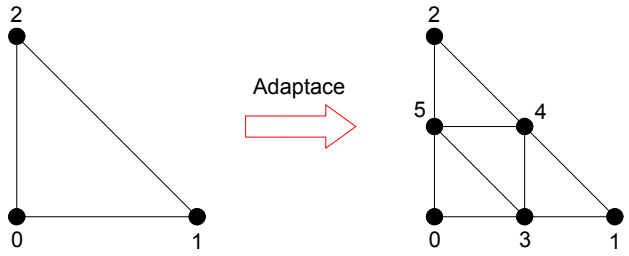
6.2.2 Dělení trojúhelníkových elementů

Princip dělení trojúhelníkových elementů je velmi obdobný tomu předchozímu. Nejprve jsou děleny vytypované elementy na menší trojúhelníkové elementy, načež následuje dotváření sítě k eliminaci všech volných uzlů. Schéma algoritmu je patrno z obrázku 26 a kód k algoritmu je uveden v příloze 8.



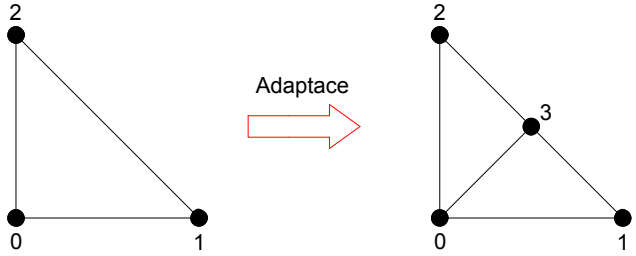
Obrázek 26: Standardní strategie dělení trojúhelníkových elementů.

Pokud je síť tvořena pouze trojúhelníkovými elementy a všechny tyto elementy jsou určeny k adaptaci, byl zde vložen algoritmus urychlující proces adaptace úlohy dělením trojúhelníků na čtyři menší trojúhelníky. Všechny nové uzly jsou umístěny do středů stran trojúhelníku. Tento proces dělení je znázorněn na obrázku 27.



Obrázek 27: Dělení trojúhelníkového elementu na 4 trojúhelníky.

Jestliže nejsou všechny elementy vyskytující se na síti trojúhelníkového tvaru nebo pokud se nemají všechny adaptovat, provede se dělení na dva menší trojúhelníky. Nový uzel je vždy umístěn do středu nejdelší strany trojúhelníku. Proces adaptace trojúhelníkového elementu je patrný z obrázku 28.

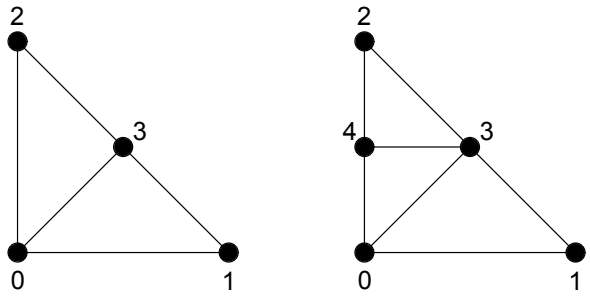


Obrázek 28: Dělení trojúhelníkového elementu na 2 trojúhelníky.

Pakliže se jedná o úlohu tlustostěnné nádoby jsou následně všechny příslušné nově vzniklé uzly ve středech hran přesunuty na skutečnou geometrickou hranici.

Po adaptaci všech elementů k tomu primárně určených je nutno dotvořit síť tak, aby se v ní nevyskytovaly žádné volné uzly. Pro dotváření čtyřúhelníků je využíváno stejného dělení, jako bylo uvedeno v kapitole 6.2.1 na obrázku 25.

Pro dotváření trojúhelníkových elementů je používáno dělení, jež je zachyceno na obrázku 29.



Obrázek 29: Implementované možnosti adaptace trojúhelníkových elementů při dosíťování k eliminaci volných uzlů.

Výstupem tohoto algoritmu jsou aktualizovaná pole nesoucí informace o hranách, na nichž jsou aplikované okrajové podmínky, pole nesoucí informace o elementech a pole se souřadnicemi jednotlivých uzlů.

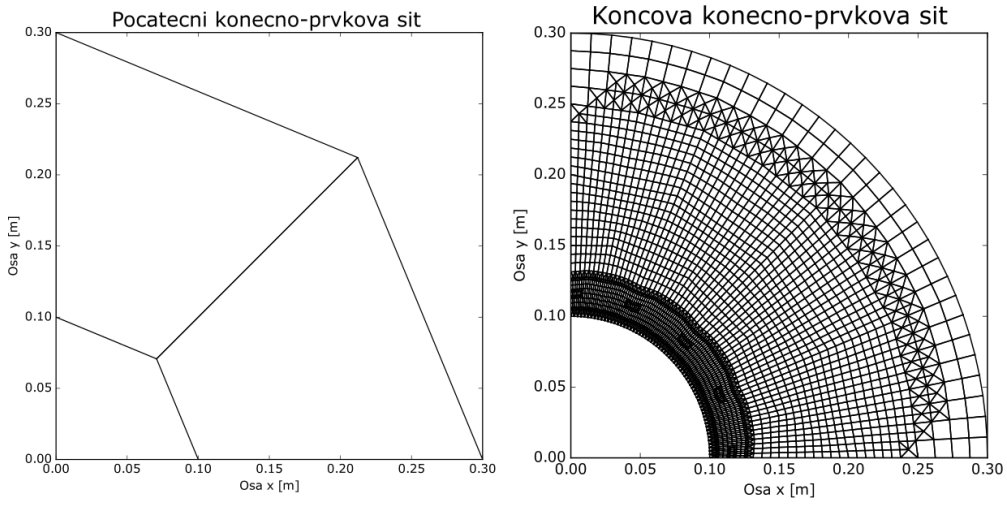
6.2.3 Proces adaptace počáteční sítě tlustostěnné nádoby Standardní strategií dělní původních elementů

Proces adaptace sítě pomocí standardní strategie dělení původních elementů bude představen na úloze tlustostěnné nádoby. Rozměry nádoby a její zatížení jsou zaznamenány v tabulce 4 a pro všechny následují výpočty zůstanou neměnné.

Vnitřní poloměr R_0 [m]	Vnější poloměr R_1[m]	Vnitřní tlak p_0 [MPa]	Vnější tlak p_1 [MPa]	Modul pružnosti v tahu E [MPa]	Poissonovo číslo μ [-]
0,1	0,3	150	0,1	210 000	0,3

Tabulka 4: Parametry a zatížení tlustostěnné nádoby.

Počáteční síť byla zhotovena ze dvou čtyřúhelníkových elementů. Hodnota maximální přípustné procentuální chyby na elementu byla vyžadována $\bar{\psi} = 5\%$.



Obrázek 30: Standardní strategie dělení - počáteční konečno-prvková síť (vlevo), síť po finální adaptaci (vpravo).

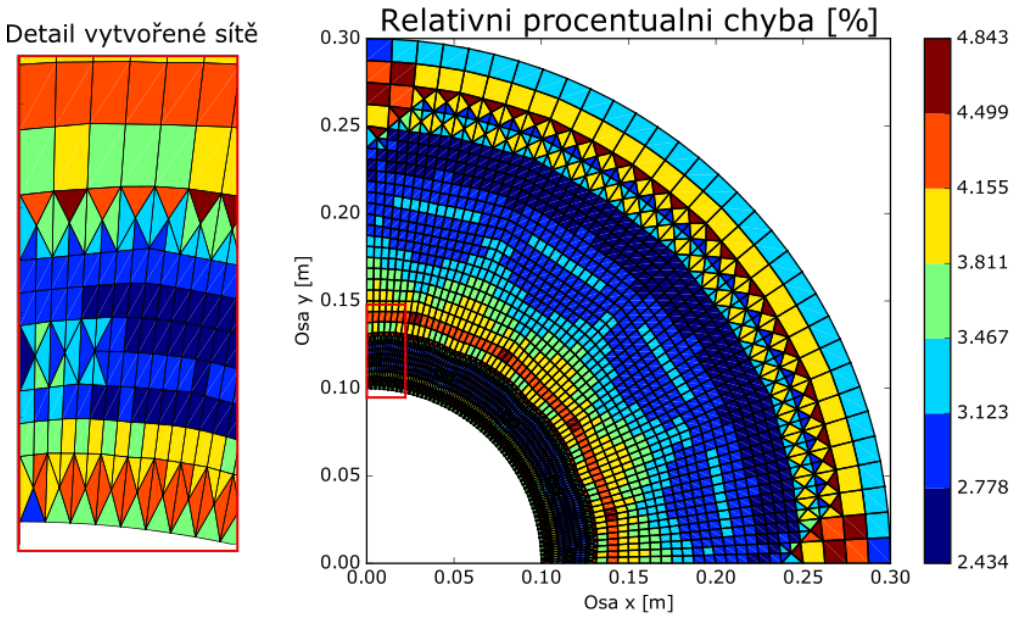
Pomocí Standardní strategie dělení bylo požadované přesnosti řešení dosaženo po šesti adaptacích. Na obrázku 30 je patrný nárůst počtu elementů z původních dvou na finální počet 3060, jež poskytuje výsledky s maximální relativní procentuální chybou 4,8 %. Lze pozorovat, že největší hustota elementů je v blízkosti vnitřního poloměru nádoby, kde se taktéž vyskytují maximální hodnoty napětí. Vytvořená finální síť je symetrická, což je s ohledem na symetričnost úlohy velmi kladný výsledek.

Dílčí výsledky jsou zaznamenány v tabulce 5. Je zde zřejmé, že hodnota relativní procentuální chyby je s každou adaptací sítě snižována, což je pozitivní faktor.

Č. adaptace	Počet elementů	Max. HMH napětí [MPa]	Max. celkové posunutí [mm]	Max. relativní chyba [MPa]
0	2	197,2	0,080	71,3
1	8	228,5	0,097	68,4
2	32	262,5	0,105	41,5
3	128	281,8	0,108	23,0
4	512	289,0	0,109	12,7
5	1788	291,1	0,109	6,8
6	3060	286,7	0,109	4,8

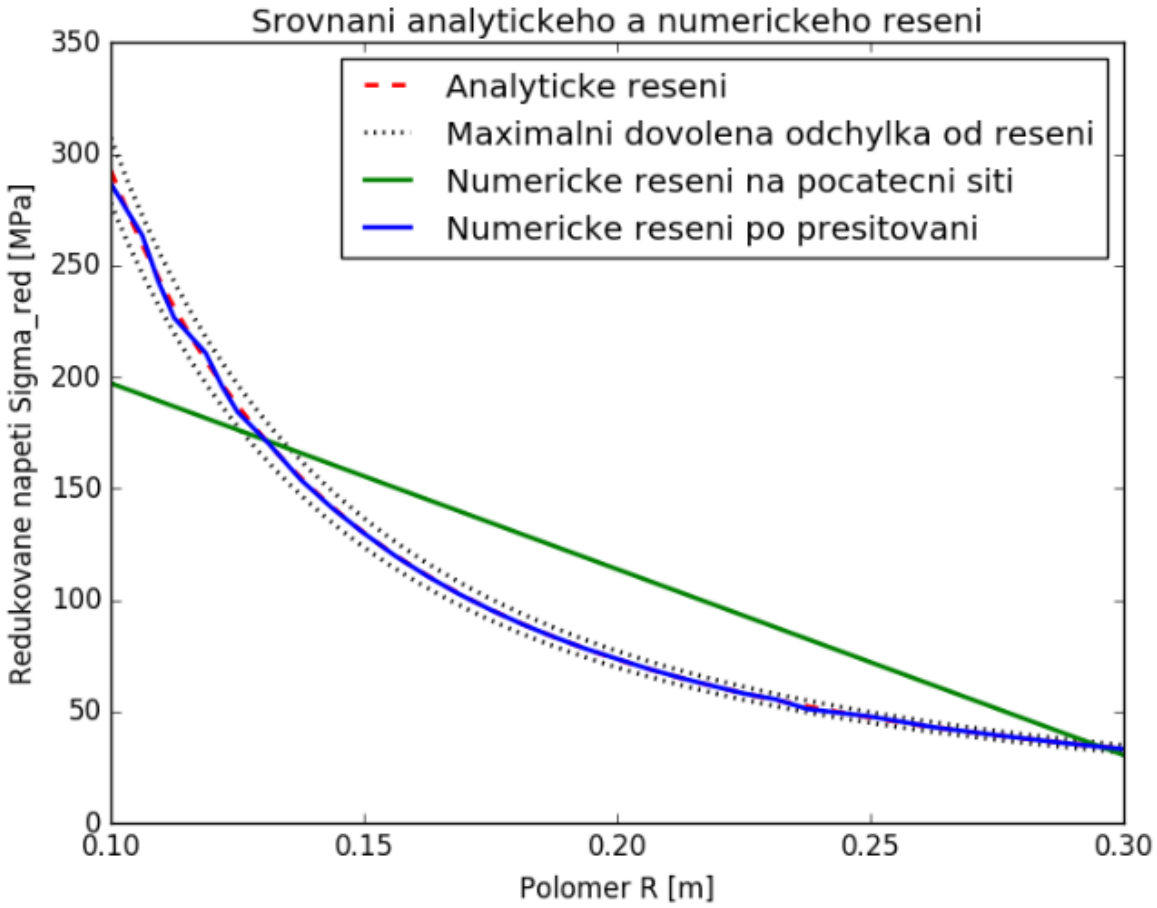
Tabulka 5: Dílčí výsledky řešení úlohy tlustostěnné nádoby s využitím standardní strategie dělení.

Zajímavým úkazem je pokles maximální hodnoty napětí při poslední adaptaci sítě. Tento fakt je způsoben změnou první řady elementů na vnitřním průměru nádoby ze čtyřúhelníků na trojúhelníky, jak je patrno z obrázku 31.



Obrázek 31: Standardní strategie dělení - rozložení relativní procentuální chyby na řešené úloze.

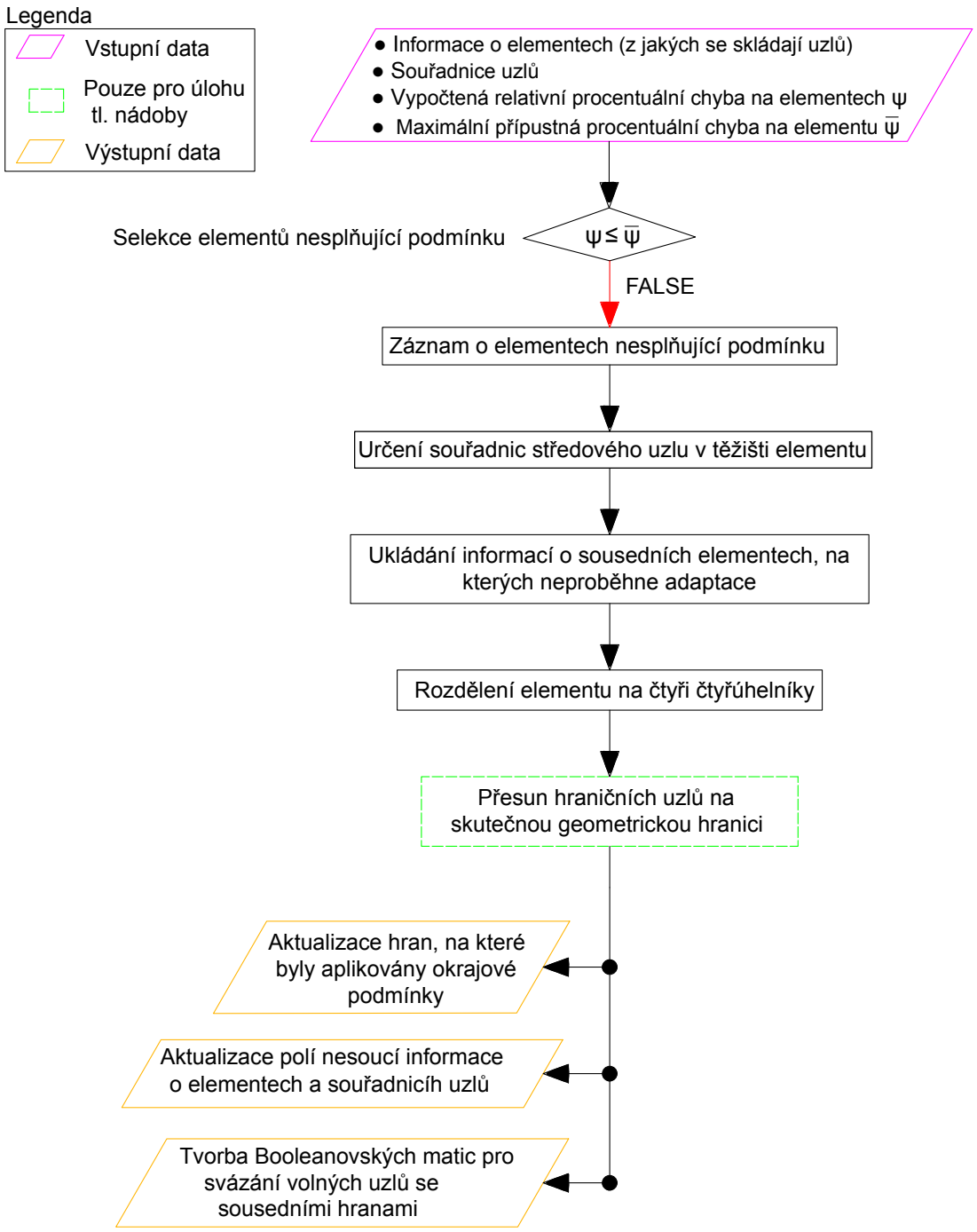
Přítomnost trojúhelníkových elementů je také vidět z průběhu redukovaného napětí dle hypotézy HMH po tloušťce nádoby, který je zachycen na obrázku32. Je zde značný pokrok mezi výsledkem z počáteční sítě, jež je reprezentován zelenou barvou, a výsledkem ze sítě adaptované. V místech, kde se vyskytují trojúhelníkové elementy, dochází k mírným odklonům redukovaného napětí od analytického řešení. Nicméně i navzdory těmto malým záchvěvům jsou všechny hodnoty napětí v požadované odchylce, jež je znázorněna čárkovanou čarou. Relativní procentuální odchylka maximálního redukovaného napětí získaného analytickým a numerickým řešením je 2,01 %.



Obrázek 32: Standardní strategie dělení - průběh redukovaného napětí dle HMH po tloušťce tlustostěnné nádoby.

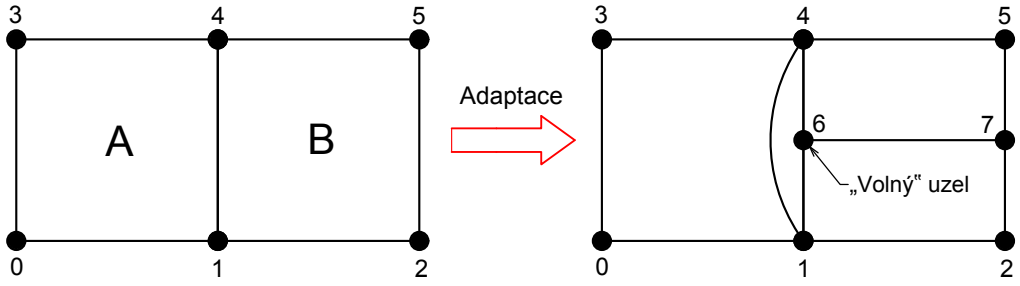
6.3 Modifikovaná strategie dělení původních elementů

Tato strategie přesíťování je určena pro sítě, které jsou tvořeny pouze čtyřúhelníkovými elementy. Pokud je tento algoritmus adaptace sítě aktivní, je potlačena adaptace trojúhelníkových elementů. Což znamená, že pokud by síť obsahovala nějaký trojúhelníkový element, nemusel by výsledek nikdy konvergovat ke správnému řešení. Schema algoritmu je naznačeno na obrázku 33 a kód s algoritmem je uveden v příloze 8.



Obrázek 33: Modifikovaná strategie dělení čtyřúhelníkových elementů.

Samotný algoritmus pracuje velice podobně jako v případě Standardní strategie dělení pro čtyřúhelníkové elementy. Změna nastává v bodě, kdy jsou již adaptovány všechny vytypované elementy. Síť může obsahovat volné uzly, které byly v předchozích případech odstraňovány dodatečným dosíťováním. Zde již k dodatečnému dosíťování nedochází, nýbrž volné uzly jsou patřičnými způsoby svazovány k sousedním elementům. K tomuto svázání je využívána *Booleanovská matice* spolu s *Lagrangeovým multiplikátorem*.



Obrázek 34: Vznik volného uzlu při adaptaci elementu **B**.

Na obrázku 34 je vlevo znázorněna počáteční síť čítající dva elementy, kde byl přiřazen každému uzlu pouze jeden stupeň volnosti. Pro jednoduchost byla síť adaptována rozbitím elementu s označením **B** na dva menší čtyřúhelníkové elementy. Touto adaptací došlo k vytvoření volného uzlu, který v obrázku 34 nese číslo **6**. Takovýto volný uzel je třeba svázat se sousední hranou tak, že jsou jeho posuvy vyjádřeny jako aritmetický průměr posuvů uzlů hrany sousedního elementu. Jinak řečeno je posuv uzlu **6** dán jako aritmetický průměr posuvů uzlů **1** a **4**, což je možné zapsat následujícím způsobem

$$u_6 = \frac{1}{2}(u_1 + u_4). \tag{67}$$

Rovnici lze upravit do tvaru, kdy na pravé straně bude figurovat nula a jednotlivé posuvy jsou seřazeny vzestupně

$$-\frac{1}{2}u_1 - \frac{1}{2}u_4 + u_6 = 0. \tag{68}$$

Tato rovnice je následně přidána jako vedlejší podmínka do systému rovnic pro řešení vektoru neznámých posuvů $\bar{\boldsymbol{u}}$ rozšířením matice tuhosti $\boldsymbol{K}$, vektoru posuvů $\bar{\boldsymbol{u}}$ a vektoru pravé strany $\boldsymbol{F}$

$$\begin{bmatrix} \boldsymbol{K} & \boldsymbol{B}^T \\ \boldsymbol{B} & \boldsymbol{0} \end{bmatrix} \begin{Bmatrix} \hat{\boldsymbol{u}} \\ \boldsymbol{\lambda} \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F} \\ \boldsymbol{0} \end{Bmatrix}, \tag{69}$$

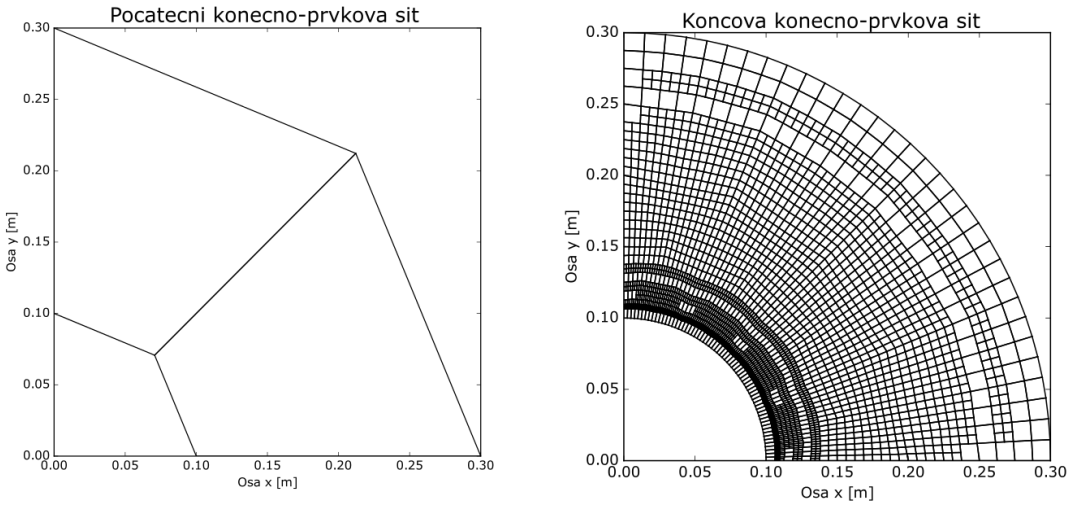
kde $\boldsymbol{B}$ představuje *Booleanovskou matici* a $\boldsymbol{\lambda}$ značí vektor *Lagrangeových multiplikátorů*, jež má fyzikální význam vnitřních sil, nutných k udržení volných uzlů na hranách sousedních elementů.

Pro případ z obrázku 34 by rozšířený systém rovnic vypadal následovně

$$\begin{bmatrix} k_{00} & k_{01} & k_{02} & k_{03} & k_{04} & k_{05} & k_{06} & k_{07} & 0 \\ k_{10} & k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} & k_{17} & -0.5 \\ k_{20} & k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} & k_{27} & 0 \\ k_{30} & k_{31} & k_{32} & k_{33} & k_{34} & k_{35} & k_{36} & k_{37} & 0 \\ k_{40} & k_{41} & k_{42} & k_{43} & k_{54} & k_{45} & k_{46} & k_{47} & -0.5 \\ k_{50} & k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} & k_{57} & 0 \\ k_{60} & k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66} & k_{67} & 1 \\ k_{70} & k_{71} & k_{72} & k_{73} & k_{74} & k_{75} & k_{76} & k_{77} & 0 \\ 0 & -0.5 & 0 & 0 & -0.5 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ \lambda \end{Bmatrix} = \begin{Bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ 0 \end{Bmatrix}. \tag{70}$$

6.3.1 Proces adaptace počáteční sítě tlustostěnné nádoby Modifikovanou strategií dělní původních elementů

Proces adaptace pomocí modifikované strategie dělení původních elementů bude představen na úloze tlustostěnné nádoby, jejíž počáteční síť je tvořena dvěma čtyřúhelníkovými elementy, obdobně jako v předchozím případě. Maximální přípustná procentuální chyba na elementu byla uvažována $\bar{\psi} = 5\%$.



Obrázek 35: Modifikovaná strategie dělení - počáteční konečno-prvková síť (vlevo), síť po finální adaptaci (vpravo).

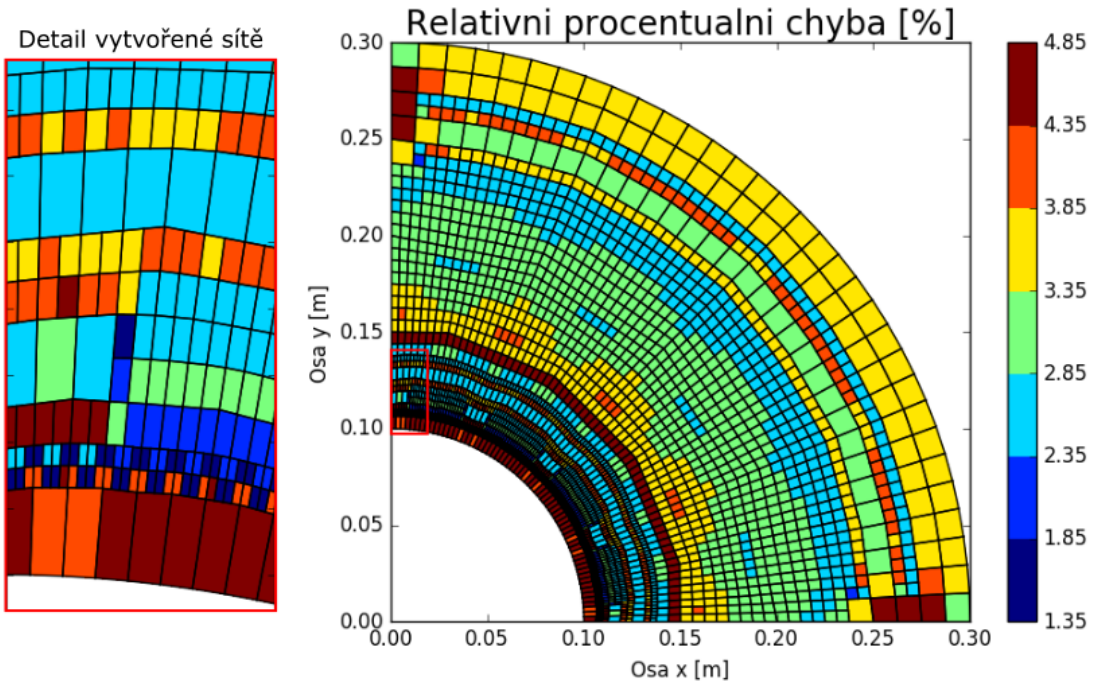
Na obrázku 35 je možno pozorovat jednak značný nárůst počtu elementů na 2846, ale také vsazenou funkci pro tlustostěnné nádoby, kterou jsou přesouvány nově vzniklé uzly ve středech hran na skutečnou geometrickou hranici. Největší hustota elementů je v oblasti vnitřního poloměru. Velmi zajímavým faktem je, že k největšímu zahuštění dochází až od druhé vrstvy elementů. Velmi pozitivním faktorem je symetričnost výsledné konečno-prvkové sítě.

Jelikož je finální síť poměrně jemná, jsou následující výsledky zobrazeny pouze na výsledné síti, přičemž výsledky z počáteční i dílčích sítí jsou zaznamenány v tabulce 6. Z této tabulky je patrné, že až do čtvrté adaptace dochází k dělení všech elementů. Po čtvrté adaptaci již na některých elementech klesne chyba pod požadovanou hodnotu a nárůst elementů se začne postupně zpomalovat. V deváté adaptaci je již na všech elementech hodnota relativní procentuální chyby pod požadovanou hranicí.

Č. adaptace	Počet elementů	Max. HMH napětí [MPa]	Max. celkové posunutí [mm]	Max. relativní chyba [MPa]
0	2	197,2	0,080	71,3
1	8	228,5	0,097	68,4
2	32	262,5	0,105	41,5
3	128	281,8	0,108	23,0
4	512	289,0	0,109	12,7
5	1658	291,1	0,109	6,8
6	2246	290,7	0,109	5,7
7	2774	290,7	0,109	5,4
8	2822	290,7	0,109	5,6
9	2846	290,7	0,109	4,8

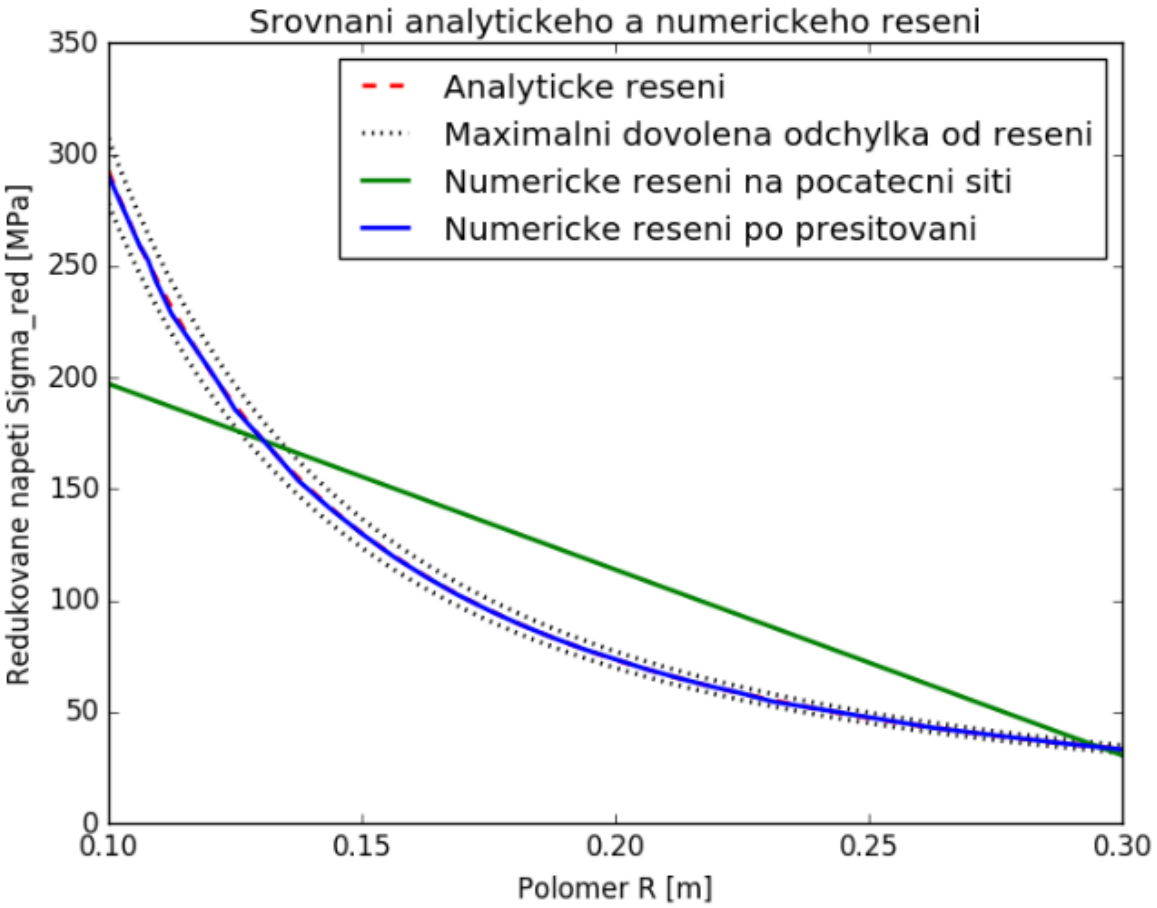
Tabulka 6: Dílčí výsledky řešení úlohy tlustostěnné nádoby s využitím modifikované strategie dělení.

Na obrázku 36 je zachyceno rozložení relativní procentuální chyby ψ na adaptované síti tlustostěnné nádoby.



Obrázek 36: Modifikovaná strategie dělení - rozložení relativní procentuální chyby na řešené úloze.

Pro lepší přehled navýšení přesnosti řešení adaptací počáteční sítě, byl vytvořen obrázek 37, jež zachycuje průběh redukovaného napětí dle hypotézy HMH po tloušťce nádoby. Zelenou barvou je vykreslen průběh napětí na počáteční síti. Jelikož byla tloušťka tlustostěnné nádoby na počátku reprezentována pouze jedním elementem, je průběh redukovaného napětí lineární. Čárkovanou čarou je vykreslena hranice vymezující požadovanou přesnost řešení. Je zde vidět, že redukované napětí získané z finální sítě téměř dokonale kopíruje analytické řešení. Relativní procentuální odchylka maximální redukovaného napětí získaného analytickým a numerickým řešením je 0,66%.



Obrázek 37: Modifikovaná strategie dělení - průběh redukovaného napětí dle HMH po tloušťce tlustostěnné nádoby.

6.4 Metody řešení vektoru neznámých uzlových posunutí $\hat{\boldsymbol{u}}$ a možnosti zavedení okrajových podmínek do řešeného systému rovnic

Jakmile je sestavena globální matice tuhosti $\boldsymbol{K}$ a globální vektor pravé strany $\boldsymbol{F}$, lze různými metodami stanovit vektor neznámých posuvů v uzlech $\hat{\boldsymbol{u}}$.

Jako intuitivní metodu lze označit osamostatnění neznámého vektoru posuvů $\hat{\boldsymbol{u}}$ převodem matice tuhosti $\boldsymbol{K}$ na pravou stranu. Aby byla úprava ekvivalentní musí se matice tuhosti $\boldsymbol{K}$ po převodu na pravou stranu invertovat. Systém rovnic je poté přetvořen do tvaru

$$\hat{\boldsymbol{u}} = \boldsymbol{K}^{-1}\boldsymbol{F}. \tag{71}$$

Proces inverze matice tuhosti $\boldsymbol{K}$ je velmi náročný, proto tento způsob výpočtu není vhodné použít a ani není jednou z variant pro řešení systému rovnic.

Již použitelnou metodou pro řešení výše zmíněného systému rovnic je metoda *LU transformace*, která spočívá v zavedení substituce matice tuhosti $\boldsymbol{K}$, jakožto součinu dolní trojúhelníkové matice $\boldsymbol{L}$ a horní trojúhelníkové matice $\boldsymbol{U}$. Systém rovnic poté přechází do tvaru

$$\boldsymbol{L}\,\boldsymbol{U}\,\hat{\boldsymbol{u}} = \boldsymbol{F}. \tag{72}$$

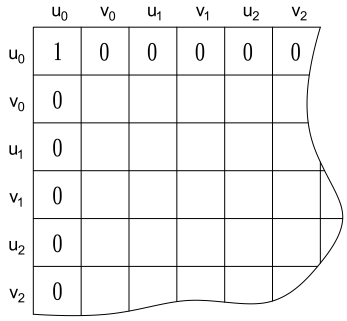
V implementovaném algoritmu lze využít pro řešení systému rovnic LU transformaci *plné matice* tuhosti, což je velmi neefektivní způsob výpočtu z hlediska využití paměti počítače. Nebo je možné s výhodou využít řídkosti matice tuhosti a LU transformaci provést s *řídkou maticí* tuhosti.

Kromě výše zmíněných explicitních řešičů existují i řešiče numerické, které pro řešení neznáme vektoru uzlových posunutí $\hat{\boldsymbol{u}}$ musí provést určité množství iterací. Tyto metody jsou výrazně efektivnější při řešení větších úloh. Ve vytvořeném MKP algoritmu je možno pro výpočet vektoru uzlových posunutí $\hat{\boldsymbol{u}}$ zvolit numerickou metodu *sdružených gradientů*.

6.4.1 LU transformace plné matice tuhosti se zavedením okrajových podmínek

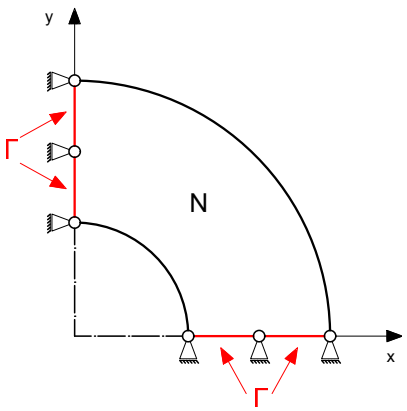
Jak již bylo zmíněno v úvodní části, pro řešení systému rovnic lze zvolit LU transformaci plné matice tuhosti. Tato metoda byla zavedena jako první a byla v programu ponechána spíše za účelem porovnání se sofistikovanějšími metodami a taktéž pro ukázku rozmanitosti možností aplikace okrajových podmínek.

Pro tuto metodu lze zvolit aplikaci okrajových podmínek formou přepisu všech pozic na řádku i ve sloupci matice tuhosti $\boldsymbol{K}$ odpovídajícímu odebranému stupni volnosti na nulu a na pozici, jež odpovídá právě tomuto stupni volnosti (například $[u_0, u_0]$) se nahraje číslo jedna. Následně je taktéž upraven vektor pravé strany $\boldsymbol{F}$, kde je na patřičnou pozici (například f_0) nahraná požadovaná hodnota posunutí.



Obrázek 38: Zavedení okrajové podmínky na posuv u_0 v globální matici tuhosti $\boldsymbol{K}$.

Druhým způsobem je úprava matice tuhosti $\boldsymbol{K}$, vektoru posunutí $\hat{\boldsymbol{u}}$ a vektoru pravé strany $\boldsymbol{F}$. Na obrázku 39 je znázorněna tlustostěnná nádoba, která byla díky její symetrii zjednodušena na pouhou čtvrtinu.



Obrázek 39: Rozdělení řešené úlohy na oblast Γ, kde jsou aplikovány okrajové podmínky, a zbylou oblast N.

Jelikož bylo uváženo tohle zjednodušení, bylo nutno aplikovat okrajové podmínky symetrie, které uzlům na hranicích Γ odebírají příslušné stupně volnosti. Zbylá oblast, kde nedojde k odebrání žádného stupně volnosti, byla označena velkým písmenem N.

Obdobné rozdělení je provedeno u matice tuhosti $\boldsymbol{K}$, vektoru posunutí $\hat{\boldsymbol{u}}$ a vektoru pravé strany $\boldsymbol{F}$

$$\begin{bmatrix} \boldsymbol{K}_{NN} & \boldsymbol{K}_{N\Gamma} \\ \boldsymbol{K}_{\Gamma N} & \boldsymbol{K}_{\Gamma\Gamma} \end{bmatrix} \begin{Bmatrix} \hat{\boldsymbol{u}}_N \\ \hat{\boldsymbol{u}}_\Gamma \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_N \\ \boldsymbol{F}_\Gamma \end{Bmatrix}, \tag{73}$$

kde jsou submaticí $\boldsymbol{K}_{NN}$ zastoupeny všechny stupně volnosti obsažené v oblasti N, submatice $\boldsymbol{K}_{\Gamma\Gamma}$ naopak obsahuje všechny stupně volnosti pro hranici Γ. Zbylé submatice obsahují informace o provázání těchto dvou výše zmíněných submatic. Podobné rozdělení je provedeno i u vektoru posunutí $\hat{\boldsymbol{u}}$ a vektoru pravé strany $\boldsymbol{F}$.

Vektor posuvů $\hat{\boldsymbol{u}}_\Gamma$ je znám, proto ho lze zapsat do pravé strany a nahradit jím vektor $\boldsymbol{F}_\Gamma$. Aby úprava byla ekvivalentní, musí se také upravit dolní submatice matice tuhosti $\boldsymbol{K}$. Soustava rovnic má následně tvar

$$\begin{bmatrix} \boldsymbol{K}_{NN} & \boldsymbol{K}_{N\Gamma} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{Bmatrix} \hat{\boldsymbol{u}}_N \\ \hat{\boldsymbol{u}}_\Gamma \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_N \\ \hat{\boldsymbol{u}}_\Gamma \end{Bmatrix}. \tag{74}$$

Matici $\boldsymbol{K}$ lze sestavit jako součet dvou dílčích matic

$$\left[\begin{pmatrix} \boldsymbol{K}_{NN} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} + \begin{pmatrix} \boldsymbol{0} & \boldsymbol{K}_{N\Gamma} \\ \boldsymbol{0} & \boldsymbol{0} \end{pmatrix} \right] \begin{Bmatrix} \hat{\boldsymbol{u}}_N \\ \hat{\boldsymbol{u}}_\Gamma \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_N \\ \hat{\boldsymbol{u}}_\Gamma \end{Bmatrix}. \tag{75}$$

Následně lze převést součin pravé dílčí matice s vektorem posuvů na pravou stranu, čímž se dostanou dva nezávislé systémy rovnic

$$\begin{bmatrix} \boldsymbol{K}_{NN} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \begin{Bmatrix} \hat{\boldsymbol{u}}_N \\ \hat{\boldsymbol{u}}_\Gamma \end{Bmatrix} = \begin{Bmatrix} \boldsymbol{F}_N - \boldsymbol{K}_{N\Gamma}\, \hat{\boldsymbol{u}}_\Gamma \\ \hat{\boldsymbol{u}}_\Gamma \end{Bmatrix}. \tag{76}$$

První systém rovnic je dán maticovým zápisem

$$\boldsymbol{K}_{NN}\, \hat{\boldsymbol{u}}_N = \boldsymbol{F}_N - \boldsymbol{K}_{N\Gamma}\, \hat{\boldsymbol{u}}_\Gamma. \tag{77}$$

jejichž řešením se stanoví všechny neznámé posuvy. Pokud jsou na hranicích Γ předepsány nulové posuvy, lze systém rovnic dále zjednodušit do podoby

$$\boldsymbol{K}_{NN}\, \hat{\boldsymbol{u}}_N = \boldsymbol{F}_N, \tag{78}$$

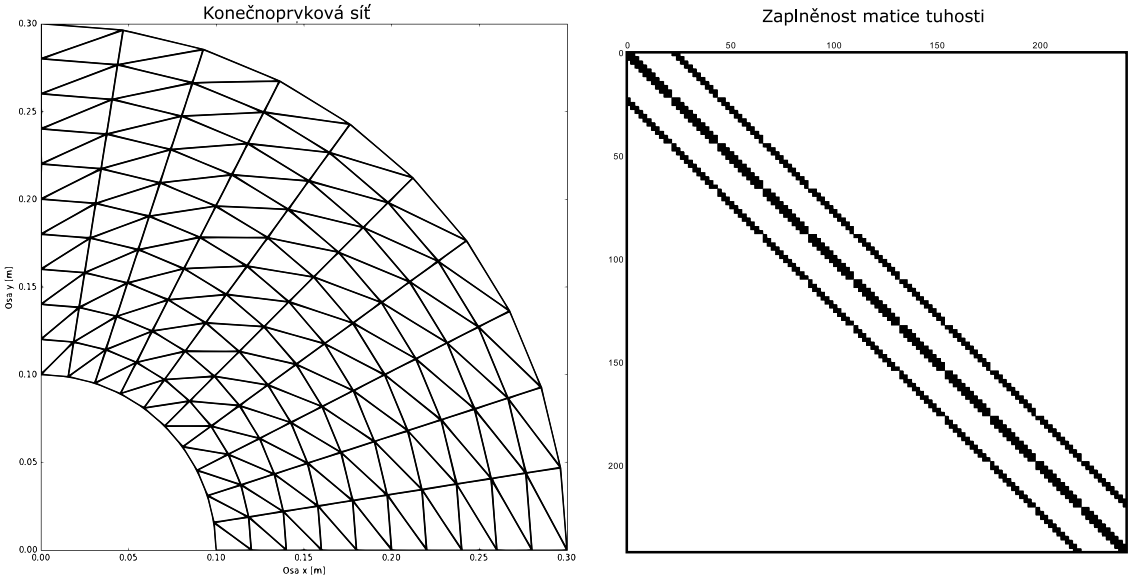
Druhý systém rovnic je splněn apriori

$$\boldsymbol{I}\ \hat{\boldsymbol{u}}_\Gamma = \hat{\boldsymbol{u}}_\Gamma\ . \tag{79}$$

Touto úpravou jsou tedy sníženy rozměry matice tuhosti $\boldsymbol{K}$, čímž také mírně poklesnou nároky na paměť počítače. Stále je zde však hranice počtu řešitelných stupňů volnosti, kdy paměť počítače již nebude dostatečně velká pro ukládání a manipulaci s maticí tuhosti $\boldsymbol{K}$.

6.4.2 LU transformace řídké matice tuhosti se zavedením okrajových podmínek

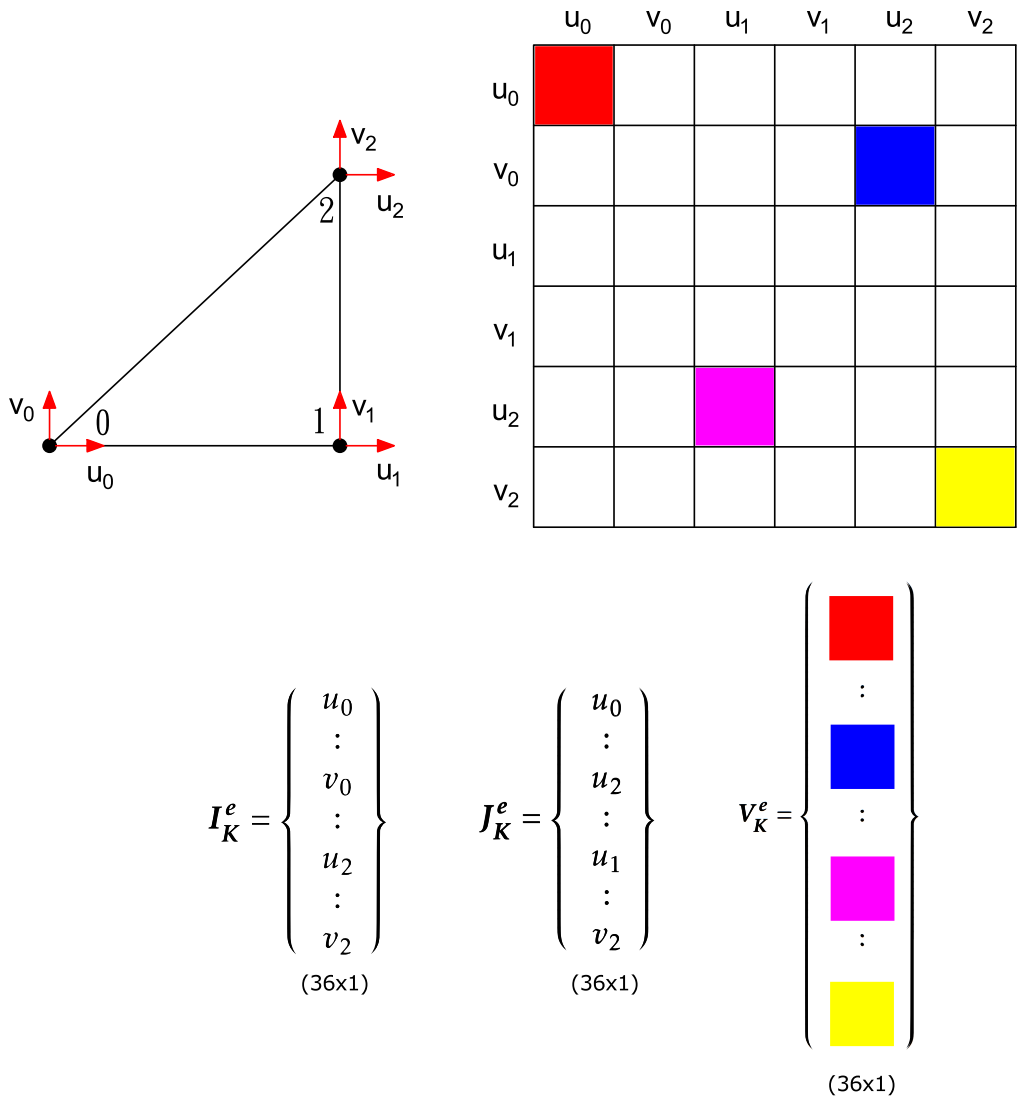
Jelikož byl MKP algoritmus vytvořen pouze pro řešení jednoduchých 2-D problémů, lze velmi efektivně využít řídkosti matice tuhosti $\boldsymbol{K}$. Na obrázku 40 je znázorněna matice tuhosti $\boldsymbol{K}$ úlohy tlustostěnné nádoby, jejíž síť tvoří 200 trojúhelníkových elementů.



Obrázek 40: Zaplněnost matice tuhosti $\boldsymbol{K}$ o rozměrech $(242x242)$.

Všechna černá pole představují příslušná místa v matici tuhosti $\boldsymbol{K}$, kde byla nahrána hodnota různá od nuly. Naopak bílým prostorem jsou znázorněna „prázdná“ místa v matici tuhosti (na těchto polích jsou nahrány nuly). Je zde vidět nepoměr mezi zaplněnými a „prázdnými“ pozicemi.

Proto je výhodnější neukládat matici tuhosti $\boldsymbol{K}$ celou, nýbrž pouze její nenulové hodnoty. Pro jejich ukládání se využívají tři vektory $\boldsymbol{I_K}$, $\boldsymbol{J_K}$ a $\boldsymbol{V_K}$. Funkce jednotlivých vektorů je znázorněna na obrázku 41.



Obrázek 41: Princip sestavení lokálních vektorů $\boldsymbol{I_K^e}$, $\boldsymbol{J_K^e}$ a $\boldsymbol{V_K^e}$ pro element trojúhelníku.

Do vektoru $\boldsymbol{V_K^e}$ jsou zaznamenávány jednotlivé hodnoty z lokální matice tuhosti elementu $\boldsymbol{k}$, přičemž jejich pozice v globální matici tuhosti $\boldsymbol{K}$ jsou zaznamenávány do vektorů $\boldsymbol{I_K^e}$ (pozice řádků) a $\boldsymbol{J_K^e}$ (pozice sloupců). Rozměr lokálních vektorů je dán druhou mocninou počtu stupňů volnosti elementu. V případě trojúhelníku z obrázku 41 je rozměr vektorů roven $(36x1)$.

Takto vytvořené záznamy o lokálních maticích tuhosti jednotlivých elementů řešené úlohy jsou poté sloučeny do globálních vektorů $\boldsymbol{I_K}$, $\boldsymbol{J_K}$ a $\boldsymbol{V_K}$.

Globální matice tuhosti $\boldsymbol{K}$ je následně složena na základě údajů z globálních vektorů $\boldsymbol{I_K}$, $\boldsymbol{J_K}$ $\boldsymbol{V_K}$ a poté je uložena ve formátu *CSC*, což je zkratka pro *Compressed Sparse Column Format*.

Okrajové podmínky, které zamezují posunutí uzlů, je možno do systému vnášet hned několika způsoby. Prvním způsobem je zkrácení matice tuhosti o odebrané stupně volnosti přesně jako v předchozí metodě. Druhou možností je navýšení tuhosti odebraných stupňů volnosti o tuhost umělou, jež zamezí posunutí uzlů v příslušných směrech. Hodnotu umělé tuhosti je nutno volit s rozvahou.

Inverzní matice tuhosti je následně stanovena užitím LU faktorizace pro řídké a čtvercové matice (v Pythonu lze využít knihovnu *SciPy*).

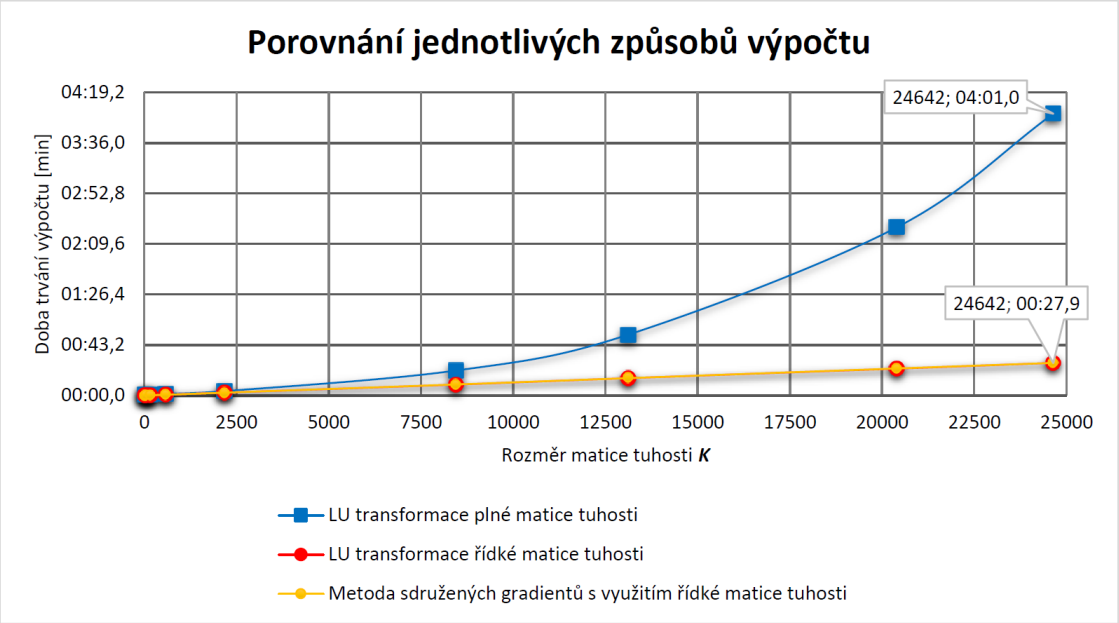
6.4.3 Metoda sdružených gradientů s využitím řídké matice tuhosti

Poslední metodou pro řešení soustavy rovnic byla zvolena a implementována metoda sdružených gradientů. Jedná se o numerickou metodu řešení, což znamená, že k dosažení požadované přesnosti výsledku je nutno provést určité množství iterací. Okrajové podmínky byly do systému vloženy obdobně jako u LU transformace. Jelikož metoda pracuje s řídkými maticemi, je velmi rychlá a efektivní.

Neboť se při aplikaci metody sdružených gradientů využívá pře-podmínění úlohy a to převrácením hodnot na hlavní diagonále matice tuhosti, nelze tuto metodu využít pro řešení úloh s využitím Modifikované strategie dělení. Při jejím využití se rozšiřuje matice tuhosti Booleanovskou maticí a na hlavní diagonále se vyskytnou nuly.

6.4.4 Porovnání efektivity jednotlivých metod řešení

Aby bylo lépe vidět efektivitu jednotlivých metod pro řešení systému rovnic, byl vytvořen obrázek 42, který sleduje poměr mezi dobou výpočtu a rozměry matice tuhosti $\boldsymbol{K}$ (což také odpovídá celkovému počtu stupňů volnosti). Všechny výpočty byly prováděny na stejném počítači a za stejných podmínek.



Obrázek 42: Porovnání efektivity uvedených způsobů řešení systému rovnic.

Na počátku lze z grafu pozorovat, že všechny uvedené metody mají přibližně stejný výpočetní výkon. Zlom nastává přibližně kolem hodnoty 4000 stupňů volnosti, kde LU transformace plné matice tuhosti začne vykazovat mocninné chování, přičemž s vyšším počtem stupňů volnosti než 24642 již použitý počítat nedokázal soustavu rovnic vyřešit. Naopak LU transformace řídké matice tuhosti a metoda sdružených gradientů řídké matice tuhosti vykazují po celou zkoumanou dobu lineární chování. Rozdíl mezi efektivními metodami a LU transformací plné matice tuhosti činí až 679,35%. Pokud by se pokračovalo v navyšování počtu stupňů volnosti, LU faktorizace řídké matice tuhosti by se taktéž stávala méně efektivní.

7 Validace vytvořeného algoritmu s komerčními MKP programy

Poslední částí práce je validace vytvořeného MKP algoritmu s vybranými komerčními MKP programy. Validace je nejprve provedena na úloze tlustostěnné nádoby a to pro dva různé stavy. Poté je řešena úloha staticky neurčité tyče, zatížené uprostřed osamělou silou.

7.1 Tlustostěnná nádoba s potlačeným přesunem nově vzniklých středových uzlů na skutečnou geometrickou hranici

Jelikož je funkce adaptace elementů v komerčním programu Marc Mentat zadávána na vytvořenou konečno-prvkovou síť, lze potlačit transformaci nově vzniklých uzlů na skutečnou geometrickou hranici. To znamená, že při použití hrubé sítě (například jako v kapitole 6.3.1) úloha nemůže konvergovat ke správnému řešení. Pro napodobení tohoto stavu byly v implementovaném algoritmu potlačeny funkce na přesun takto vzniklých hraničních uzlů. Jelikož se začíná s velice hrubou sítí, bylo možno poměrně dobře sledovat a porovnávat chování všech algoritmů pro adaptaci sítě.

Aby bylo srovnání co nejvíce adekvátní byly v obou programech nastaveny stejné počáteční podmínky, které jsou zaneseny v tabulce 7.

Typ elementů	Počet elementů na počáteční síti	Požadovaná přesnost	Maximální počet adaptací
Čtyř-uzlový prvek	2	10 %	6

Tabulka 7: Počáteční podmínky pro validaci algoritmu na úloze tlustostěnné nádoby s potlačeným přesunem nově vzniklých středových uzlů na skutečnou geometrickou hranici.

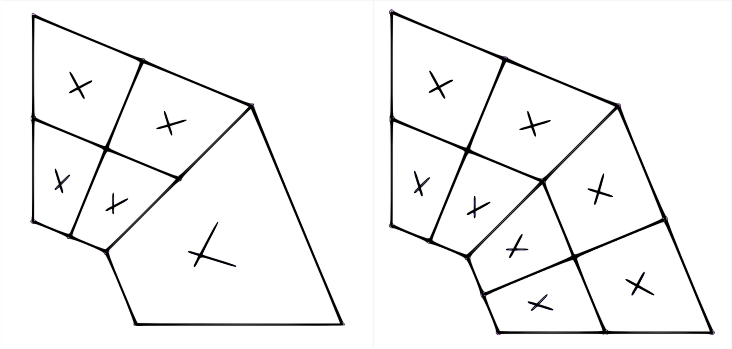
Jelikož jsou v takto nastavené úloze obsažena místa se singularitami, jsou požadavky na přesnost a také maximální počet dovolených adaptací přizpůsobeny tomuto faktu. Při zvolení větší přesnosti řešení a zvětšení maximálního počtu adaptací by vytvořená finální síť nebyla zřetelná (síť v oblasti singularit by byla značně zahuštěná).

Tak jako v předchozích případech, byly dílčí výsledky obou programů zaneseny do tabulky 8.

	Č. adaptace	0	1	2	3	4	5	6
Marc Mentat	Počet elementů	2	5	8	20	44	92	206
	Max. HMH napětí [MPa]	122,4	171,2	171,5	221,1	284,3	354,1	458,9
	Max. celkové posunutí [mm]	0,080	0,096	0,098	0,104	0,106	0,109	0,110
Standard. strategie dělení	Počet elementů	2	8	32	128	564	1144	1788
	Max. HMH napětí [MPa]	197,2	231,6	304,2	391,5	505,6	650,9	828,9
	Max. celkové posunutí [mm]	0,080	0,094	0,103	0,107	0,109	0,110	0,110
Modifik. strategie dělení	Počet elementů	2	8	32	128	422	734	1094
	Max. HMH napětí [MPa]	197,2	231,6	304,2	391,5	505,9	652,4	831,2
	Max. celkové posunutí [mm]	0,080	0,094	0,103	0,107	0,109	0,110	0,110

Tabulka 8: Dílčí výsledky porovnání vlastního algoritmu s programem Marc Mentat pro úlohu tlustostěnné nádoby s potlačením funkce přesunu nově vzniklých středových uzlů na skutečnou geometrickou hranici.

Z tabulky 8 je patrné, že adaptace v programu Marc Mentat probíhá pomaleji než ve vlastním MKP algoritmu. Tento fakt je zřejmý hlavně v prvních krocích, kdy Marc Mentat první adaptací provede dělení pouze na jednom ze dvou elementů, načež druhý element rozdělí až při druhé adaptaci, jak je vidět na obrázku 43.



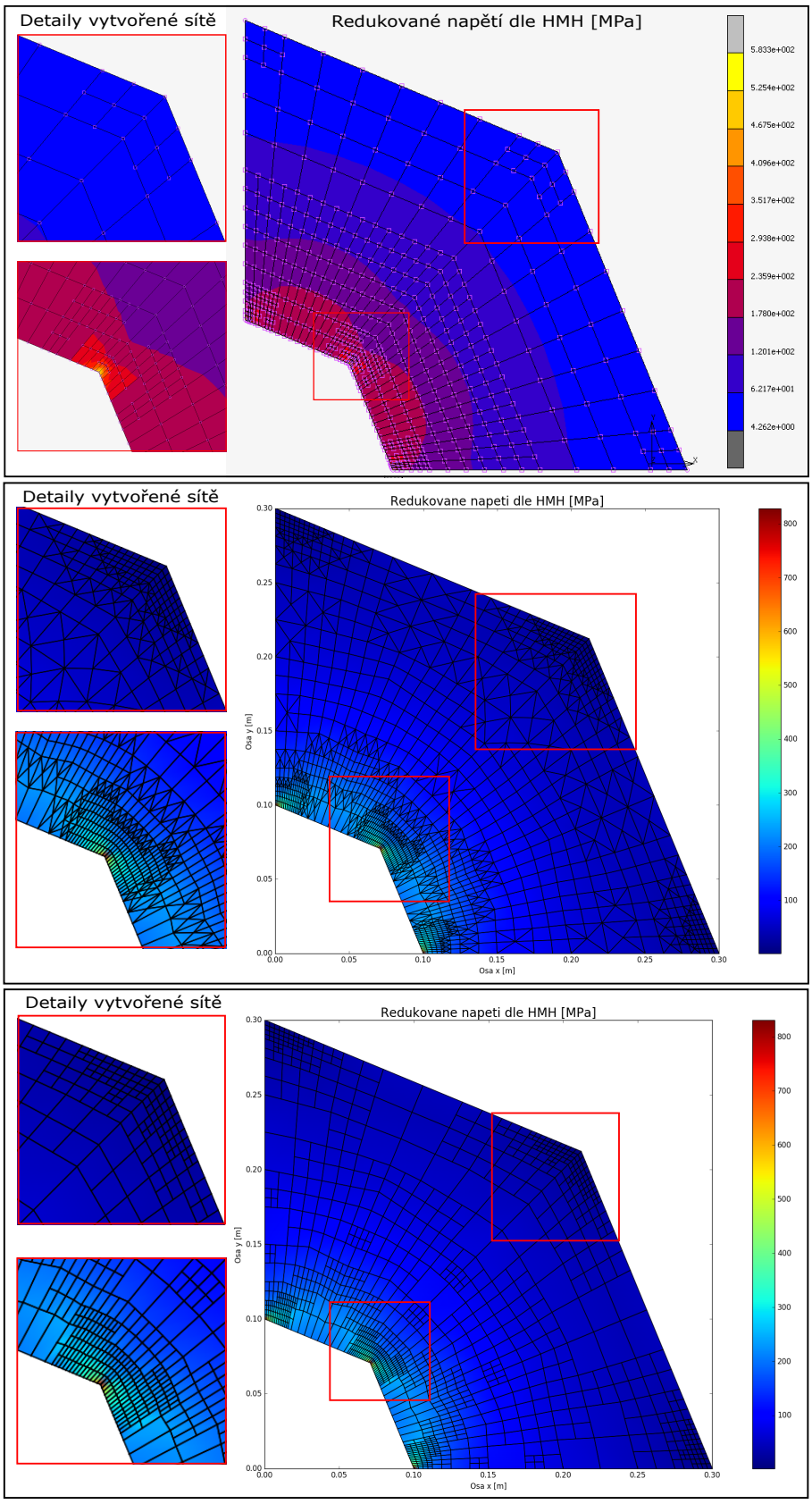
Obrázek 43: Adaptace sítě tl. nádoby komerčním programem Marc Mentat - adaptace v prvním kroku vlevo, adaptace v druhém krou vpravo.

Touto skutečností je markantně ovlivněn rozdíl v celkovém počtu elementů mezi vlastním MKP algoritmem a komerčním programem Marc Mentat. V posledním adaptačním kroku je rozdíl v počtu elementů pro Modifikovanou strategii dělení více než pětinásobný a pro Standardní strategii dělení je rozdíl téměř devítinásobný. Výsledné sítě lze vidět na obrázku 44, kde je také zachyceno rozložení redukovaného napětí dle hypotézy HMH.

Jelikož se jedná o úlohu obsahující singularity, je zřejmé, že zjemňováním sítě v okolí těchto singularit se získané napětí neustále navyšuje. Neboť jsou sítě z vytvořeného MKP algoritmu jemnější v okolí singulárních míst, dosahují maximální hodnoty redukovaného napětí vyšších hodnot než v případě programu Marc Mentat.

Z obrázku 44 je zřejmé, že obě implementované strategie dělení vytváří velmi podobné sítě (pokud se jedná pouze o tvorbu čtyřúhelníkových elementů). Rozdíl v počtech elementů mezi implementovanými strategiemi je způsoben nutností dotváření sítě dělením čtyřúhelníkových elementů na různý počet trojúhelníků v případě Standardní strategie dělení.

Neboť se jedná o symetrickou úlohu, je velmi pozitivním faktem, že všechny vytvořené sítě jsou symetrické.



Obrázek 44: Průběh napětí na finálních sítích - Marc Mentat (horní síť), Standardní strategie dělení (prostřední síť), Modifikovaná strategie dělení (dolní síť).

7.2 Tlustostěnná nádoba s aktivním přesunem nově vzniklých středových uzlů na skutečnou geometrickou hranici

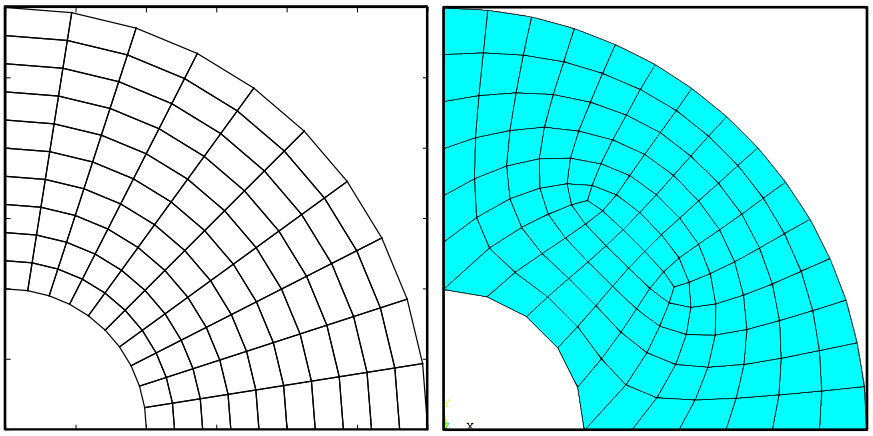
Stejná úloha byla poté řešena s jemnější počáteční sítí, což by mělo urychlit konvergenci úlohy. Byly opět opět aktivovány funkce na přesun nově vzniklých uzlů ve středech hran na skutečnou geometrickou hranici. Tato funkce je v programu ANSYS APDL nastavena jako výchozí, proto nebyla zařazena do minulého srovnání.

Tak jako v předchozím případě, i zde byla kladena snaha o stejné počáteční podmínky pro všechny programy. Jelikož jsou podmínky pro tvorbu počáteční sítě v programu ANSYS APDL omezeny, byla vytvořena počáteční síť čítají přibližně stejný počet elementů jako v ostatních programech. Tyto podmínky jsou zřejmé z tabulky 9.

Typ elementů	Počet elementů na počáteční síti	Požadovaná přesnost	Maximální počet adaptací
Čtyř-uzlový prvek	100	5 %	10

Tabulka 9: Počáteční podmínky pro validaci algoritmu na úloze tlustostěnné nádoby.

Počáteční sítě pro všechny programy jsou zřejmé z obrázku 45. Je vidět, že počáteční síť v programu ANSYS APDL je na vnitřním poloměru nádoby horší kvality, než je tomu u ostatních programů.



Obrázek 45: Počáteční síť použitá u vlastního algoritmu a taktéž v programu Marc Mentat (vlevo), počáteční síť vytvořená v programu ANSYS APDL (vpravo).

Dílčí výsledky jsou zaznamenány v tabulce 10. Jelikož všechny programy či použité strategie adaptace dosáhli požadované přesnosti za různý počet adaptací, byla tomu tabulka přizpůsobena. Poslední záznam pro každý uvedený program či strategii adaptace je zároveň adaptací, kdy bylo dosaženo požadované přesnosti řešení.

	Č. adaptace	0	1	2	3	5	7
Marc Mentat	Počet elementů	100	190	400	784	-	-
	Max. HMH napětí [MPa]	264,9	277,2	291,9	310,5	-	-
	Max. celkové posunutí [mm]	0,108	0,109	0,109	0,109	-	-
ANSYS APDL	Počet elementů	115	230	438	-	-	-
	Max. HMH napětí [MPa]	282,4	280,6	284,0	-	-	-
	Max. celkové posunutí [mm]	0,109	0,109	0,109	-	-	-
Standard. strategie dělení	Počet elementů	100	400	1376	1916	2116	2242
	Max. HMH napětí [MPa]	286,0	289,5	291,1	288,5	288,5	288,4
	Max. celkové posunutí [mm]	0,108	0,109	0,109	0,109	0,109	0,109
Modifik. strategie dělení	Počet elementů	100	400	1294	1468	1522	-
	Max. HMH napětí [MPa]	286,0	289,5	291,1	290,4	290,4	-
	Max. celkové posunutí [mm]	0,108	0,109	0,109	0,109	0,109	-

Tabulka 10: Dílčí výsledky porovnání vlastního algoritmu s programem Marc Mentat a ANSYS APDL pro úlohu tlustostěnné nádoby.

Z tabulky 10 je vidět, že oběma komerčními MKP programy byla ukončena adaptace velmi brzy a navíc s poměrně malým počtem elementů.

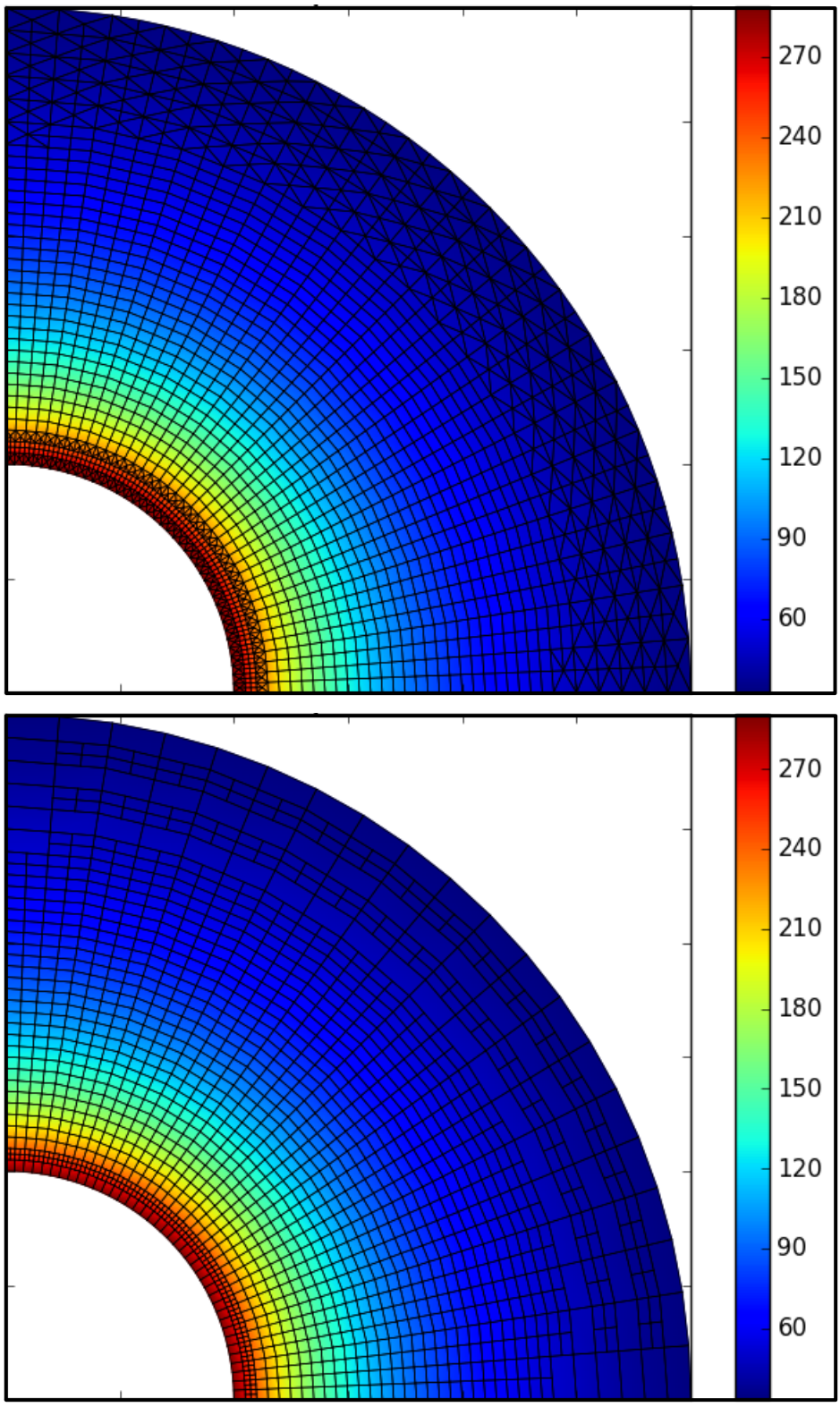
Jako nejlepší se při adaptaci projevil ANSYS APDL, který dokázal vytvořit síť zaručující požadovanou přesnost řešení a to již za dvě adaptace s výsledným počtem elementů 438. Jeho algoritmus pracuje zcela odlišně než algoritmy ostatních programů. Nedochází k dělení původních elementů, nýbrž je tvořena zcela nová síť, jak již bylo zmíněno v kapitole 5.2.2. Maximální redukované napětí je rozdílné od analytického o 2,93 %.

Druhou pozici v efektivnosti adaptace sítě obdržel program Marc Mentat. Požadované přesnosti řešení dosáhl za tři adaptace a výsledná síť sestávala z 784 elementů. To je o 79 % více elementů než v případě programu ANSYS APDL. Je zajímavé, že v poslední adaptaci došlo k zjemnění pouze některých elementů na vnitřním poloměru nádoby. Tento fakt se projevil ve výsledné hodnotě maximálního napětí, jež se odklonilo od analytického řešení o 6,12 %.

Nejlepší z implementovaných adaptivních algoritmů se v tomto případě ukázala Modifikovaná strategie dělení elementů, jež dosáhla přesného řešení za 5 adaptací s finálním počtem elementů 1522. Při porovnání s nejefektivnějším programem ANSYS APDL bylo k dosažení požadované přesnosti řešení nutno použít o 248 % více elementů. Lze tedy konstatovat, že adaptační algoritmus programu ANSYS APDL je velice efektivní. Příznivější shody dosáhla Modifikovaná strategie dělení s programem Marc Mentat, kde je výsledný počet elementů vyšší o 94 %. Na druhou stranu bylo touto strategií dělení dosaženo nejlepší shody s analyticky stanovenou maximální hodnotou redukovaného napětí, kde relativní procentuální chyba dosáhla hodnoty 0,75 %.

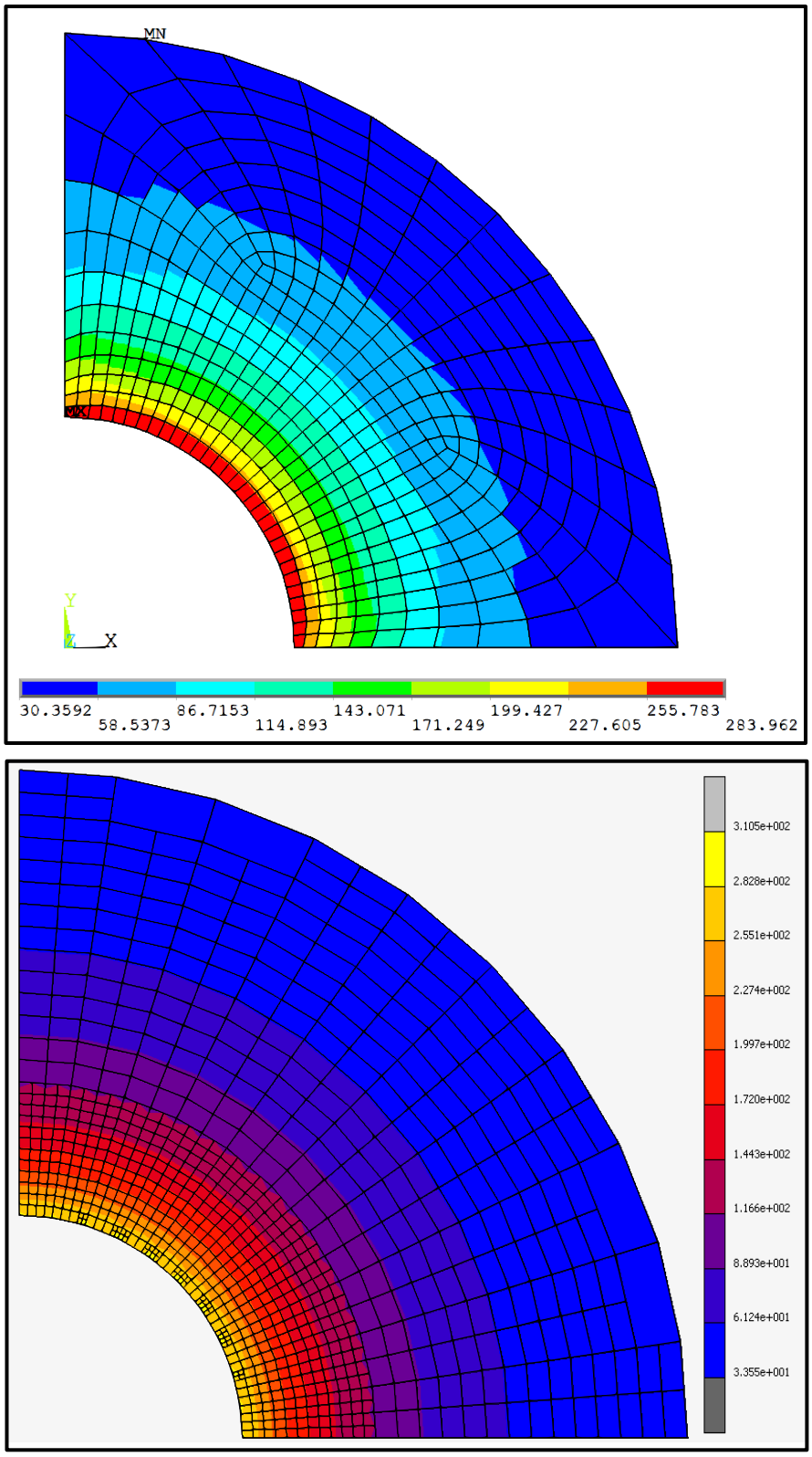
Nejméně efektivním implementovaným algoritmem se ukázala Standardní strategie dělení, jež potřebovala k dosažení požadované přesnosti řešení 7 adaptací a výsledná síť čítala 2242 elementů. Největší slabina této strategie dělení je nutnost dotváření sítě k eliminaci volných uzlů dělením patřičných elementů na různý počet trojúhelníků. Tímto dotvářením sítě se může stát, že nově vzniklé trojúhelníky neposkytnou lepší řešení a naopak u nich dojde k navýšení relativní procentuální chyby, načež se poté musí v dalším kole adaptovat.

Na následujícím obrázku 46 je zachycena finální síť získaná užitím Standardní a Modifikované strategie dělení.



Obrázek 46: Průběh redukovaného napětí [MPa] na finálních sítích - Standardní strategie dělení (horní síť), Modifikovaná strategie dělení (dolní síť).

Další obrázek 47 zachycuje finální sítě získané komerčními programy ANSYS APDL a Marc Mentat.



Obrázek 47: Průběh redukovaného napětí [MPa] na finálních sítích - ANSYS APDL (horní síť), Marc Mentat (dolní síť).

7.3 Vetknutá tyč zatížena bodovou silou

Jako poslední byla validace provedena na příkladu staticky neurčité obdélníkové tyče, vetknuté na obou stranách a uprostřed zatížené bodovou silou. Na obrázku 48 je znázorněna tyč, na kterou bylo díky symetrii úlohy uplatněno zjednodušení.

Obrázek 48: Schéma úlohy staticky neurčité tyče zatížené bodovou silou.

Jedná se o úlohu obsahující singularity, v jejichž blízkosti by mělo docházet k zjemnění sítě. Parametry tyče a zatížení jsou zaneseny v tabulce 11.

Délka tyče L [m]	Výška profilu H [m]	Tloušťka profilu t [m]	Zatěžující síla F [N]	Modul pružnosti v tahu E [MPa]	Poissonovo číslo μ [-]
0,2	0,03	0,01	8 000	210 000	0,3

Tabulka 11: Parametry a zatížení staticky neurčité tyče.

Tak jako u předchozích úloh jsou i zde počáteční podmínky zaznamenány v tabulce 12.

Typ elementů	Počet elementů na počáteční síti	Požadovaná přesnost	Maximální počet adaptací
Čtyř-uzlový prvek	20	20 %	4

Tabulka 12: Počáteční podmínky pro validaci algoritmu na úloze vetknuté tyče.

Počáteční síť byla vytvořena tak, že čítala 20 čtyřúhelníkových elementů. Požadovaná přesnost řešení zde byla stanovena na hodnotu $\psi = 20$ %, aby síť po finální adaptaci byla dobře rozpoznatelná.

Dílčí výsledky jsou zaznamenány v tabulce 13. Neboť všechny programy či použité strategie adaptace dosáhli požadované přesnosti za různý počet adaptací, byla tomu tabulka přizpůsobena. Poslední záznam pro každý uvedený program či strategii adaptace je zároveň adaptací, kdy bylo dosaženo požadované přesnosti řešení.

	Č. adaptace	0	1	2	3	4
Marc Mentat	Počet elementů	20	44	95	-	-
	Max. HMH napětí [MPa]	432,6	534,9	664,5	-	-
	Max. celkové posunutí [mm]	0,996	1,105	1,146	-	-
ANSYS APDL	Počet elementů	115	408	-	-	-
	Max. HMH napětí [MPa]	463,8	768,2	-	-	-
	Max. celkové posunutí [mm]	0,996	1,196	-	-	-
Standard. strategie dělení	Počet elementů	20	80	326	1200	2273
	Max. HMH napětí [MPa]	565,8	678,3	915,7	1529,8	2857,0
	Max. celkové posunutí [mm]	0,996	1,143	1,184	1,195	1,200
Modifik. strategie dělení	Počet elementů	20	80	311	854	1514
	Max. HMH napětí [MPa]	565,8	678,3	914,4	1540,7	2861,3
	Max. celkové posunutí [mm]	0,996	1,143	1,182	1,199	1,203

Tabulka 13: Dílčí výsledky porovnání vlastního algoritmu s programem Marc Mentat a ANSYS APDL pro úlohu staticky neurčité tyče.

Při bližším pohledu do tabulky 13 je opět zřejmé, že komerční programy ukončili adaptaci sítě před maximální stanovenou hranicí. Nejrychleji byla adaptace dokončena programem ANSYS APDL, kterým bylo poskytnuto řešení s požadovanou přesností jednou adaptací počáteční sítě, přičemž byl navýšen počet elementů na 408.

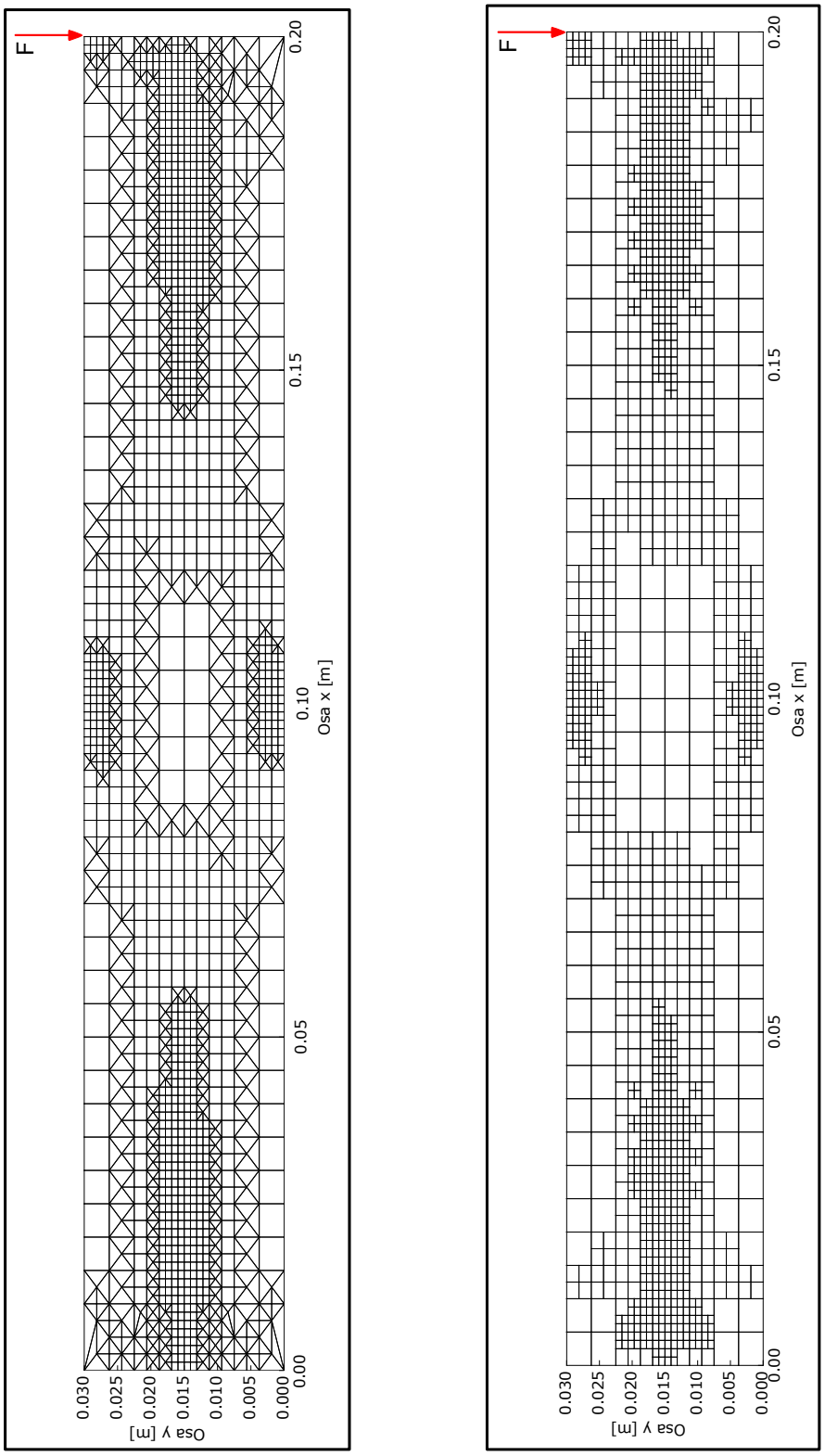
Velmi hrubá síť v porovnání s programem ANSYS APDL byla poskytnuta programem Marc Mentat, který potřeboval k dosažení požadované přesnosti řešení dvě adaptace, přičemž výsledná síť čítala 95 elementů. Což je v porovnání s programem ANSYS APDL o 329 % méně elementů.

Všechny implementované strategie dělení využili maximální možný počet adaptací. Modifikovanou strategií dělení byla vytvořena finální síť čítající 1514 elementů, což je v porovnání s programem ANSYS APDL o 271 % více elementů. Takto velké zjemnění se také projevilo na hodnotě maximálního redukovaného napětí, jež je situováno pod zatěžující silou. Užitím Standardní strategie dělení byla vytvořena finální síť, která sestávala z 2273 elementů.

Všechny užité programy či strategie dělení se s mírnou odchylkou shodují v hodnotě maximálního celkového posunutí. Největší odchylka je mezi Modifikovanou strategií dělení elementů a programem Marc Mentat, kdy relativní procentuální chyba dosahuje hodnoty 5 %.

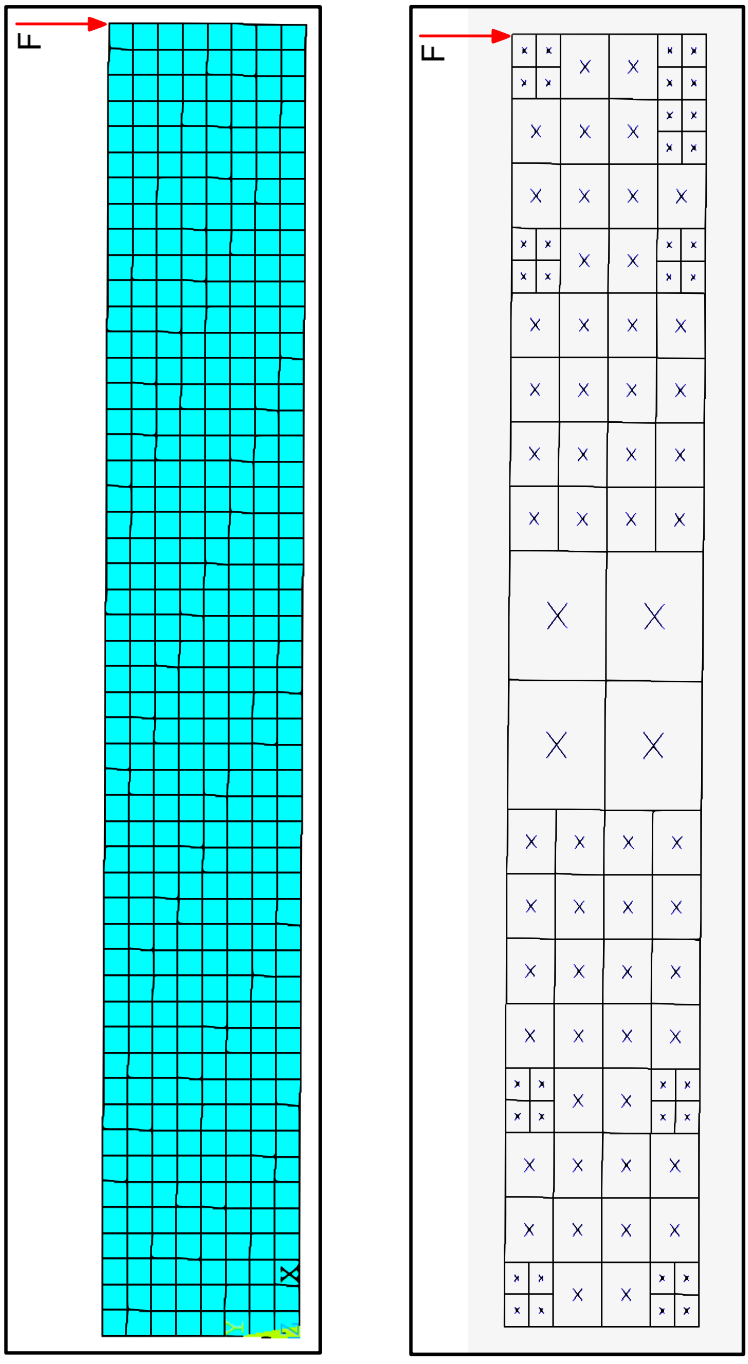
Díky singulárnímu bodu pod působící silou je průběh napětí zcela nevhodný k prezentaci vytvořených sítí. Proto zde budou vytvořené sítě prezentovány bez dodatečného vykreslení jakéhokoli průběhu vypočtené fyzikální veličiny.

Na obrázku 49 jsou zachyceny finální sítě, které byly vytvořeny vlastními adaptivními algoritmy.



Obrázek 49: Finální síť vytvořená Standardní strategií dělení (vlevo), finální síť vytvořená Modifikovanou strategií dělení (vpravo).

Na obrázku 50 jsou zachyceny finální sítě, které byly vytvořeny komerčními programy. Je zřejmé, že programem ANSYS APDL byla síť zjemněna zcela kompletně a tím bylo dosaženo požadované přesnosti řešení. Naopak programem Marc Mentat byla síť zjemňována lokálně a je zde vidět určitá shoda s vlastními adaptačními algoritmy.



Obrázek 50: Finální síť vytvořená komerčním programem ANSYS APDL (vlevo), finální síť vytvořená komerčním programem Marc Mentat (vpravo).

8 Závěr

Cílem diplomové práce bylo vytvořit vlastní MKP algoritmus pro řešení dvoudimenzionálních úloh zahrnující podporu adaptivních technik. Techniky řešiteli umožňují vytvářet poměrně hrubou počáteční síť a dle stanovených kritérií následně síť upravit tak, aby poskytla řešení s požadovanou přesností.

Na úvod byly představeny elementární rovnice teorie pružnosti, jež jsou základním kamenem celé problematiky metody konečných prvků. Také byly definovány případy rovinné úlohy, tj rovinná deformace a rovinná napjatost. Na závěr kapitoly byla zavedena substituce, která umožňovala formálně stejný zápis materiálové matice tuhosti pro oba stavy rovinné úlohy.

V další kapitole byla stručně zmíněna teorie metody konečných prvků. Větší pozornost byla věnována referenčním prvkům pro rovinnou úlohu, jež jsou obsaženy v implementovaném algoritmu.

Následující kapitola byla zaměřena na adaptivní techniky v metodě konečných prvků. Byly zde uvedeny dvě základní metody používané pro zpřesnění výsledků. První představenou metodou byla h-verze, jež se později stala primárním bodem zájmu při implementaci vlastního adaptivního algoritmu. Zmíněna byla taktéž p-verze adaptace sítě. Poté byl definován aposteriorní odhad chyby autorů Zienkiewicze a Zhua. Pro implementaci vlastního algoritmu byla využita varianta stanovení energetické normy na základě rozdílu vyhlazeného a integrovaného napětí.

V praktické části práce byl nejprve představen konečno-prvkový algoritmus jako celek. V algoritmu byly vytvořeny dvě strategie dělení elementů.

První byl popsán algoritmus adaptace sítě s názvem Standardní strategie dělení elementů. Tento algoritmus byl implementován převážně pro adaptaci smíšené sítě (síť tvořená z trojúhelníkových a čtyřúhelníkových elementů). Algoritmus byl představen na úloze tlustostěnné nádoby jejíž počáteční síť byla tvořena dvěma čtyřúhelníkovými elementy. Požadované přesnosti bylo dosaženo po šesti adaptacích a finální síť čítala 3060 elementů. Průběh redukovaného napětí po tloušťce tlustostěnné nádoby obsahoval mírné výkyvy, jež byly způsobeny přítomností trojúhelníkových elementů.

Druhým implementovaným algoritmem byla Modifikovaná strategie dělení elementů, jež je určena pouze pro adaptaci sítě tvořené čistě čtyřúhelníkovými elementy. Adaptivní algoritmus byl představen na stejné úloze jako předchozí. Poža-

dované přesnosti řešení bylo dosaženo po 9 adaptacích a finální síť čítala 2846 elementů. Na průběhu redukovaného napětí po tloušťce tlustostěnné nádoby bylo znát, že síť je tvořena čistě čtyřúhelníkovými elementy, neboť byl průběh bez výrazných výchylek.

V předposlední kapitole byly zmíněny způsoby řešení systému rovnic a taktéž možnosti aplikace okrajových podmínek. Zajímavou a velmi důležitou byla podkapitola věnující se řídkým maticím, které v implementovaném programu dovolují řešiteli počítat rozměrnější úlohy.

Na závěr práce byla realizována validace vytvořeného MKP algoritmu s komerčními programy ANSYS APDL a Marc Mentat.

První validace byla provedena pouze s programem Marc Mentat a to na úloze tlustostěnné nádoby s potlačeným přesunem nově vzniklých uzlů na skutečnou geometrickou hranici. Na této úloze bylo možné poměrně dobře sledovat odlišnosti mezi vlastím a komerčním algoritmem pro adaptaci sítě. Velmi zajímavým bodem bylo zjištění, že algoritmus programu Marc Mentat v případě výskytu singulárních bodů pracuje pomaleji něž algoritmy vlastní. Tento rozdíl byl viditelný jednak ve finálním počtu elementů, ale také v dílčích výsledcích.

Dále byla validace opět uskutečněna na tlustostěnné nádobě, ovšem s jemnější počáteční síti a taktéž aktivovaným přesunem nově vzniklých uzlů na skutečnou geometrickou hranici. Zde byl do porovnání již zařazen i program ANSYS APDL. Nejlépe při srovnání s komerčními programy dopadla Modifikovaná strategie dělení, jež poskytla řešení s požadovanou přesností za 5 adaptací, což je o 3 více než ANSYS APDL a o 2 více než Marc Mentat. Výsledná síť byla tvořena z 1522 elementů. Při porovnání s nejefektivnějším programem ANSYS APDL je to o 248 % elementů více. Tento markantní rozdíl je způsoben odlišnou strategií adaptace sítě programu ANSYS APDL. Při porovnání s programem Marc Mentat, který je implementované strategii bližší, byl rozdíl v počtech elementů 94 %. Nejméně pozitivní výsledek jak v počtu elementů tak v nutných adaptacích sítě byl prokázán Standardní strategií dělení elementů. Výsledná síť čítala 2242 elementů, které byly vytvořeny až sedmou adaptací počáteční sítě.

Poslední komparační úlohou byla staticky neurčitá tyč, vetknutá na obou svých koncích a uprostřed zatížená osamělou silou. Porovnání dopadlo podobně tomu předchozímu. U obou komerčních programů byla adaptace ukončena dříve než u implementovaných strategií dělení. Nejhrubší síť byla vytvořena po dvou adapta-

cích programem Marc Mentat. V síti bylo obsaženo pouze 95 elementů. Programem ANSYS APDL byla finální síť zhotovena pouze jednou adaptací a bylo v ní zahrnuto 408 elementů. Dle očekávání byla z pohledu implementovaných strategií nejefektivnější Modifikovaná strategie dělení, kterou byla vytvořena síť čítající 1514 elementů. Kvůli nutnosti eliminace volných uzlů byla Standardní strategií dělení složena síť z ještě většího počtu elementů (2273).

Závěrem je možné konstatovat, že obě implementované strategie jsou schopny provádět cílenou (lokální) adaptaci sítě a poskytnout řešiteli výsledky s požadovanou přesností. Jejich efektivita je však v porovnání s komerčními programy poměrně nízká. U obou implementovaných strategií dělení je potřeba k dosažení požadované přesnosti řešení více adaptací, přičemž ve finále je vytvořena jemnější síť než v případě obou komerčních programů.

Další kroky při vývoji implementovaného algoritmu by tedy měly nejprve směřovat do modifikace aposteriorního odhadu chyby nad elementy. S tímto krokem je úzce spjata implementace dodatečného algoritmu na spojování elementů, u kterých je naopak vykazováno extrémně malých relativních procentuálních chyb. Tímto dodatečným algoritmem by mohlo být poskytnuto rovnoměrnějšího rozložení relativní procentuální chyby. Vhodná by byla také optimalizace prohledávání souřadnic uzlů pro zamezení vytváření uzlů duplikátních. Zde byla použita velmi jednoduchá smyčka, která však při větším množství uzlů dokáže výpočet velmi zpomalit.

Poděkování

Tímto bych rád poděkoval vedoucímu práce panu Ing. Alexandru Markopoulosovi, Ph.D. za odborné vedení, cenné rady a věnovaný čas. Poděkování patří taktéž mé rodině, přítelkyni a přátelům za jejich podporu.

Použitá literatura

[1] LENERT, Jiří. *Pružnost a pevnost II.* Ostrava: VŠB-Technická univerzita, 1998. ISBN 80-7078-572-1.

[2] HÖSCHL, Cyril. *Nelineární problémy mechaniky deformovaných těles.* Praha: Dům techniky ČSVTS Praha, 1988.

[3] HÖSCHL, Cyril. *Pružnost a pevnost ve strojnictví.* Praha: SNTL, 1971.

[4] FUXA, Jan a Ludmila ADÁMKOVÁ. *Sbírka příkladů z pružnosti a pevnosti II.* Ostrava: VŠB - Technická univerzita Ostrava, 2008. ISBN 978-80-248-1288-5.

[5] BITTNAR, Zdeněk a Jiří ŠEJNOHA. *Numerické metody mechaniky 1.* Praha: ČVUT, 1992. ISBN 80-010-0855-X.

[6] BITTNAR, Zdeněk a Jiří ŠEJNOHA. *Numerické metody mechaniky 2.* Praha: ČVUT, 1992. ISBN 80-010-0901-7.

[7] LENERT, Jiří. *Úvod do metody konečných prvků.* Ostrava: VŠB-Technická univerzita, 1999. ISBN 80-707-8686-8.

[8] FUSEK, Martin a Jaroslav ROJÍČEK. *Metoda konečných prvků I.* Ostrava: VŠB-Technická univerzita, 2013. ISBN 978-80-248-3023-0.

[9] PETRUŠKA, Jindřich. *Počítačové metody mechaniky II* [online]. Vysoké učení technické v Brně [cit. 2017-02-22]. Dostupné z: `http://www.kvm.tul.cz/getFile/id:2499`

[10] SMITH, I. M. a D. V. GRIFFITHS. *Programming the finite element method.* 2nd ed. New York: Wiley, c1988. ISBN 047191553X.

[11] LOGAN, Daryl L. *A first course in the finite element method.* 5th ed. Stamford, CT: Cengage Learning, c2012. ISBN 04-956-6825-7.

[12] AKIN, J. E. *Finite element analysis with error estimators: an introduction to the FEM and adaptive error analysis for engineering students.* 5th ed. Boston: Elsevier/Butterworth-Heinemann, 2005. ISBN 07-506-6722-2.

[13] ZIENKIEWICZ, O. C. a Robert L. TAYLOR. *The finite element method.* 5th ed. Boston: Butterworth-Heinemann, 2000. ISBN 07-506-5050-8.

[14] BHATTI, M. Asghar. *Advanced topics in finite element analysis of structures: with Mathematica and MATLAB computations.* Hoboken, N.J.: John Wiley, c2006. ISBN 978-0-471-64807-9.

[15] *MCS Software: Volume A: Theory and User Information* [online]. U.S.A: MSC Software Corporation, 2014 [cit. 2017-02-11]. Dostupné z: https://simcompanion.mscsoftware.com/infocenter/index?page=content&id=DOC10568&cat=MARC_DOCUMENTATION_2014&actp=LIST

[16] ANSYS, Inc. *ANSYS Release 15.0 Documentation.* 2014. Dostupné v elektronické podobě jako součást softwaru ANSYS 15.0.

[17] Python. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2017 [cit. 2017-04-08]. Dostupné z: https://cs.wikipedia.org/wiki/Python

[18] NumPy. *NumPy* [online]. numFOCUS, 2017 [cit. 2017-04-08]. Dostupné z: http://www.numpy.org/

[19] Matplotlib: *Matplotlib* [online]. 2017 [cit. 2017-04-08]. Dostupné z: http://matplotlib.org/

Příloha A - Standardní strategie dělení čtyřúhelníkových elementů

```
import numpy as np

def Remesh(coordinates,Rem_elements,elements,Pos,tvar,edge0,edge1,edge2,
    edge3,R1,R0,Hrana,Tl_nadoba):
     Minus = np.array(([]),dtype=np.int32)
     for j in range(Rem_elements.shape[0]):
         if Rem_elements[j,-1] > -1:
#Zjisteni pozic hran vybraneho elementu v matici Hrana + serazeni dle
    cisla hrany, ktere je obsazeno v poslednim sloupci matice Hrana
             Pozice = np.zeros((4,2),dtype=np.int32)
             cnt = 0
             for i, k in enumerate(Hrana):
                 if Hrana[i,2] == Pos[j]:
                     Pozice[cnt,0] = i
                     Pozice[cnt,1] = Hrana[i,3]
                     cnt += 1

             col = 1
             Pozice = Pozice[np.argsort(Pozice[:,col])]
             Pozice = np.delete(Pozice,1,1)
#=====================================================================
#Detekce sousednich elementu
             Soused = np.zeros((4,5),dtype=np.int32)
             for i in range(4):
                 if Pozice[i,0] == 0:
                     cnt = 1
                     while Hrana[Pozice[i,0],0] == Hrana[Pozice[i,0]+cnt
    ,0]:
                         if np.all(Hrana[Pozice[i,0],:2] == Hrana[Pozice[i
    ,0]+cnt,:2]):
                             Soused[i,:3] = Hrana[Pozice[i,0]+cnt,:3]
                             Soused[i,3]  = Pozice[i,0]+cnt
                             Soused[i,-1] = i
                             break
                         else:
                             cnt += 1

```

```
34                  if Pozice[i,0] == Hrana.shape[0]-1:
35                      cnt = 1
36                      while Hrana[Pozice[i,0],0] == Hrana[Pozice[i,0]-cnt
    ,0]:
37                          if np.all(Hrana[Pozice[i,0],:2] == Hrana[Pozice[i
    ,0]-cnt,:2]):
38                              Soused[i,:3] = Hrana[Pozice[i,0]-cnt,:3]
39                              Soused[i,3]  = Pozice[i,0]-cnt
40                              Soused[i,-1] = i
41                              break
42                          else:
43                              cnt += 1
44
45                  if 0 < Pozice[i,0] < Hrana.shape[0]-1:
46                      for k in range(2):
47                          if k == 0:
48                              cnt = 1
49
50                              while Hrana[Pozice[i,0],0] == Hrana[Pozice[i
    ,0]-cnt,0]:
51                                  if np.all(Hrana[Pozice[i,0],:2] == Hrana[
    Pozice[i,0]-cnt,:2]):
52                                      Soused[i,:3] = Hrana[Pozice[i,0]-cnt
    ,:3]
53                                      Soused[i,3]  = Pozice[i,0]-cnt
54                                      Soused[i,-1] = i
55                                      break
56                                  else:
57                                      cnt += 1
58
59                          if k == 1:
60                              cnt = 1
61                              while Hrana[Pozice[i,0],0] == Hrana[Pozice[i
    ,0]+cnt,0]:
62                                  if np.all(Hrana[Pozice[i,0],:2] == Hrana[
    Pozice[i,0]+cnt,:2]):
63                                      Soused[i,:3] = Hrana[Pozice[i,0]+cnt
    ,:3]
64                                      Soused[i,3]  = Pozice[i,0]+cnt
65                                      Soused[i,-1] = i
```

```
66                                      break
67                                  elif Pozice[i,0] + cnt == Hrana.shape
    [0]-1:
68                                      break
69                                  else:
70                                      cnt += 1
71
72                  for i in range(Soused.shape[0]):
73                      if elements[Soused[i,2],-1] == -1:
74                          Minus = np.append(Minus,[j],0)
75
76      Minus = np.unique(Minus)
77
78      for j in range(Minus.shape[0]):
79          Rem_elements = np.delete(Rem_elements,Minus[Minus.shape[0]-(j+1)
    ],0)
80          Pos = np.delete(Pos,Minus[Minus.shape[0]-(j+1)],0)
81
82
83      ones = np.ones(3)
84      Adaptace_na_trojuhelniky = np.array(([]),dtype=np.int32)
85
86 #=============================================================================
87 #Urceni teziste ctyruzloveho prvku
88      for j in range(Rem_elements.shape[0]):
89          if Rem_elements[j,-1] >= 0:
90                  Area_C = 0.0
91                  Vaha_x = 0.0
92                  Vaha_y = 0.0
93                  for i in range(2):
94                      ie = Rem_elements[j,:]
95                      ie_t = [ie[0],ie[i+1],ie[i+2]]
96                      xi = coordinates[ie_t,0]
97                      yi = coordinates[ie_t,1]
98                      Ah_ = np.array([ones,xi,yi]).transpose()
99                      Area_h = np.linalg.det(Ah_)*0.5
100                     x_tt = (1/3)*(xi[0] + xi[1] + xi[2])
101                     y_tt = (1/3)*(yi[0] + yi[1] + yi[2])
102
103                     Vaha_x += Area_h*x_tt
```

```
104                     Vaha_y += Area_h*y_tt
105                     Area_C += Area_h
106 
107                 x_T = Vaha_x/Area_C
108                 y_T = Vaha_y/Area_C
109 
110                 coordinates = np.append(coordinates,[[x_T,y_T]],0)
111                 Teziste = coordinates.shape[0]-1
112 
113 #======================================================================
114 #Zjisteni pozic hran vybraneho elementu v matici Hrana + serazeni dle
    cisla hrany, ktere je obsazeno v poslednim sloupci matice Hrana
115                 Pozice = np.zeros((4,2),dtype=np.int32)
116                 cnt = 0
117                 for i, k in enumerate(Hrana):
118                     if Hrana[i,2] == Pos[j]:
119                         Pozice[cnt,0] = i
120                         Pozice[cnt,1] = Hrana[i,3]
121                         cnt += 1
122 
123                 col = 1
124                 Pozice = Pozice[np.argsort(Pozice[:,col])]
125                 Pozice = np.delete(Pozice,1,1)
126 #======================================================================
127 #Detekce sousednich elementu
128                 Soused = np.zeros((4,5),dtype=np.int32)
129                 for i in range(4):
130                     if Pozice[i,0] == 0:
131                         cnt = 1
132                         while Hrana[Pozice[i,0],0] == Hrana[Pozice[i,0]+
    cnt,0]:
133                             if np.all(Hrana[Pozice[i,0],:2] == Hrana[
    Pozice[i,0]+cnt,:2]):
134                                 Soused[i,:3] = Hrana[Pozice[i,0]+cnt,:3]
135                                 Soused[i,3]  = Pozice[i,0]+cnt
136                                 Soused[i,-1] = i
137                                 break
138                             else:
139                                 cnt += 1
140 
```

```
141                     if Pozice[i,0] == Hrana.shape[0]-1:
142                         cnt = 1
143                         while Hrana[Pozice[i,0],0] == Hrana[Pozice[i,0]-
    cnt,0]:
144                             if np.all(Hrana[Pozice[i,0],:2] == Hrana[
    Pozice[i,0]-cnt,:2]):
145                                 Soused[i,:3] = Hrana[Pozice[i,0]-cnt,:3]
146                                 Soused[i,3]  = Pozice[i,0]-cnt
147                                 Soused[i,-1] = i
148                                 break
149                             else:
150                                 cnt += 1
151
152                     if 0 < Pozice[i,0] < Hrana.shape[0]-1:
153                         for k in range(2):
154                             if k == 0:
155                                 cnt = 1
156
157                                 while Hrana[Pozice[i,0],0] == Hrana[
    Pozice[i,0]-cnt,0]:
158                                     if np.all(Hrana[Pozice[i,0],:2] ==
    Hrana[Pozice[i,0]-cnt,:2]):
159                                         Soused[i,:3] = Hrana[Pozice[i,0]-
    cnt,:3]
160                                         Soused[i,3]  = Pozice[i,0]-cnt
161                                         Soused[i,-1] = i
162                                         break
163                                     else:
164                                         cnt += 1
165
166                             if k == 1:
167                                 cnt = 1
168                                 while Hrana[Pozice[i,0],0] == Hrana[
    Pozice[i,0]+cnt,0]:
169                                     if np.all(Hrana[Pozice[i,0],:2] ==
    Hrana[Pozice[i,0]+cnt,:2]):
170                                         Soused[i,:3] = Hrana[Pozice[i,0]+
    cnt,:3]
171                                         Soused[i,3]  = Pozice[i,0]+cnt
172                                         Soused[i,-1] = i
```

```
173                                         break
174                                     elif Pozice[i,0] + cnt == Hrana.shape
    [0]-1:
175                                         break
176                                     else:
177                                         cnt += 1
178
179 #Pokud hrana elementu nema souseda, je stale zapotrebi oznacit cislo
    hrany kvuli naslednemu serazeni
180                     if np.all(Soused[i,:] == 0):
181                         Soused[i,-1] = i
182
183 #Serazeni sousedu
184                 col = 4
185                 Soused = Soused[np.argsort(Soused[:,col])]
186
187 #Pokud doslo k pripadu zminenem vyse, je zapotrebi po serazeni cislo
    hrany prepsat nulou pro spravnou funkci algoritmu
188                 for i in range(4):
189                     if np.all(Soused[i,:4] == 0):
190                         Soused[i,-1] = 0
191
192 #=========================================================================
193 #Kontrola zda sousedni element neni trojuhelnik a zda se bude sousedni
    element presitovavat
194                 for i in range(4):
195                     if np.all(Soused[i,:] == 0):
196                         continue
197                     else:
198 #Pokud je sousedni element trojuhelnik, provede se presitovani pomoci
    rozdeleni na 4 trojuhelniky
199                         if elements[Soused[i,2],-1] == -1:
200                             tvar[j] = 1
201
202 #Pokud je sousedni element urcen k presitovani, neni potreba se s nim
    zaobirat
203                         elif np.any(Pos[:] == Soused[i,2]):
204                             if tvar[j] == 1:
205                                 print('Pass')
206                                 break
```

```
207
208                                 else:
209                                     Soused[i,:] = 0
210                             else:
211                                 continue
212
213 #=====================================================================
214 #Pokud je ctyruzlovy prvek zdeformovany rozdeli se na 4 trojuhelniky
215                 if tvar[j] == 1:
216                     print(Rem_elements[j])
217                     Adaptace_na_trojuhelniky = np.append(
    Adaptace_na_trojuhelniky,Pos[j],0)
218
219 #=====================================================================
220 #Pokud je ctyruzlovy prvek nezdeformovany, lze rozdelit na 4 mensi
    ctyruhelniky
221                 else:
222                     T = np.zeros((4),dtype=np.int32)
223                     for i in range(4):
224 #                         print(Hrana[Pozice[i,0],0],Hrana[Pozice[i,0],1])
225                         x_T = 0.5*(coordinates[Hrana[Pozice[i,0],0],0]+
    coordinates[Hrana[Pozice[i,0],1],0])
226                         y_T = 0.5*(coordinates[Hrana[Pozice[i,0],0],1]+
    coordinates[Hrana[Pozice[i,0],1],1])
227
228                         for k in range(coordinates.shape[0]):
229                             if coordinates[k,0] == x_T:
230                                 if coordinates[k,1] == y_T:
231                                     T[i] = k
232                                     break
233
234                                 elif k == coordinates.shape[0]-1:
235                                     coordinates = np.append(coordinates
    ,[[x_T,y_T]],0)
236                                     T[i] = coordinates.shape[0]-1
237                                     break
238
239                             elif k == coordinates.shape[0]-1:
240                                 coordinates = np.append(coordinates,[[x_T
    ,y_T]],0)
```

```
241                                 T[i] = coordinates.shape[0]-1
242
243
244                         if np.all(Soused[i,:] == 0):
245                             continue
246                         else:
247                             Hrana[Soused[i,3],-1] = T[i]
248
249                     elements[Pos[j],:] = [[Rem_elements[j,0],T[0],Teziste
    ,T[3]]]
250                     elements = np.append(elements,[[T[0],Rem_elements[j
    ,1],T[1],Teziste]],0)
251                     elements = np.append(elements,[[Teziste,T[1],
    Rem_elements[j,2],T[2]]],0)
252                     elements = np.append(elements,[[T[3],Teziste,T[2],
    Rem_elements[j,3]]],0)
253 #=====================================================================
254 #Aktualizace edge0-4
255                     for i in range(edge0.shape[0]):
256                         for k in range(3):
257                             Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
258                             if np.any(edge0[i,:] == Vybr_par[0]):
259                                 if np.any(edge0[i,:] == Vybr_par[1]):
260                                     edge0[i,0] = Vybr_par[0]
261                                     edge0[i,1] = T[k]
262                                     edge0 = np.append(edge0,[[Vybr_par
    [1],T[k]]],0)
263
264 #Presun noveho uzlu na vnitrni polomer nadoby
265                                     if Tl_nadoba == True:
266                                         p = np.array([coordinates[T[k
    ],0],coordinates[T[k],1]])
267                                         c = R0/(p[0]**2+p[1]**2)**0.5
268                                         coordinates[T[k],0] = p[0]*c
269                                         coordinates[T[k],1] = p[1]*c
270
271                         Vybr_par = np.array([Rem_elements[j,3],
    Rem_elements[j,0]])
272                         if np.any(edge0[i,:] == Vybr_par[0]):
```

```
273                             if np.any(edge0[i,:] == Vybr_par[1]):
274                                 edge0[i,0] = Vybr_par[0]
275                                 edge0[i,1] = T[3]
276                                 edge0 = np.append(edge0,[[Vybr_par[1],T
    [3]]],0)
277
278 #Presun noveho uzlu na vnitrni polomer nadoby
279                                 if Tl_nadoba == True:
280                                     p = np.array([coordinates[T[3],0],
    coordinates[T[3],1]])
281                                     c = R0/(p[0]**2+p[1]**2)**0.5
282                                     coordinates[T[3],0] = p[0]*c
283                                     coordinates[T[3],1] = p[1]*c
284
285                     for i in range(edge1.shape[0]):
286                         for k in range(3):
287                             Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
288                             if np.any(edge1[i,:] == Vybr_par[0]):
289                                 if np.any(edge1[i,:] == Vybr_par[1]):
290                                     edge1[i,0] = Vybr_par[0]
291                                     edge1[i,1] = T[k]
292                                     edge1 = np.append(edge1,[[Vybr_par
    [1],T[k]]],0)
293
294                         Vybr_par = np.array([Rem_elements[j,3],
    Rem_elements[j,0]])
295                         if np.any(edge1[i,:] == Vybr_par[0]):
296                             if np.any(edge1[i,:] == Vybr_par[1]):
297                                 edge1[i,0] = Vybr_par[0]
298                                 edge1[i,1] = T[3]
299                                 edge1 = np.append(edge1,[[Vybr_par[1],T
    [3]]],0)
300
301
302                     for i in range(edge2.shape[0]):
303                         for k in range(3):
304                             Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
305                             if np.any(edge2[i,:] == Vybr_par[0]):
```

```
306                                 if np.any(edge2[i,:] == Vybr_par[1]):
307                                     edge2[i,0] = Vybr_par[0]
308                                     edge2[i,1] = T[k]
309                                     edge2 = np.append(edge2,[[Vybr_par
    [1],T[k]]],0)
310
311 #Presun noveho uzlu na vnejsi polomer nadoby
312                                     if Tl_nadoba == True:
313                                         p = np.array([coordinates[T[k
    ],0],coordinates[T[k],1]])
314                                         c = R1/(p[0]**2+p[1]**2)**0.5
315                                         coordinates[T[k],0] = p[0]*c
316                                         coordinates[T[k],1] = p[1]*c
317
318                             Vybr_par = np.array([Rem_elements[j,3],
    Rem_elements[j,0]])
319                             if np.any(edge2[i,:] == Vybr_par[0]):
320                                 if np.any(edge2[i,:] == Vybr_par[1]):
321                                     edge2[i,0] = Vybr_par[0]
322                                     edge2[i,1] = T[3]
323                                     edge2 = np.append(edge2,[[Vybr_par[1],T
    [3]]],0)
324
325 #Presun noveho uzlu na vnejsi polomer nadoby
326                                     if Tl_nadoba == True:
327                                         p = np.array([coordinates[T[3],0],
    coordinates[T[3],1]])
328                                         c = R1/(p[0]**2+p[1]**2)**0.5
329                                         coordinates[T[3],0] = p[0]*c
330                                         coordinates[T[3],1] = p[1]*c
331
332                         for i in range(edge3.shape[0]):
333                             for k in range(3):
334                                 Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
335                                 if np.any(edge3[i,:] == Vybr_par[0]):
336                                     if np.any(edge3[i,:] == Vybr_par[1]):
337                                         edge3[i,0] = Vybr_par[0]
338                                         edge3[i,1] = T[k]
339                                         edge3 = np.append(edge3,[[Vybr_par
```

```
    [1],T[k]]],0)
340
341                         Vybr_par = np.array([Rem_elements[j,3],
    Rem_elements[j,0]])
342                         if np.any(edge3[i,:] == Vybr_par[0]):
343                             if np.any(edge3[i,:] == Vybr_par[1]):
344                                 edge3[i,0] = Vybr_par[0]
345                                 edge3[i,1] = T[3]
346                                 edge3 = np.append(edge3,[[Vybr_par[1],T
    [3]]],0)
347
348
349 #=====================================================================
350 #Dositovani sousednich elementu, ktere nebyly urceny k presitovani, ale
    na jejich hranach vznikli nove uzly
351
352 #=====================================================================
353 #Uprava matice Hrana tak, aby v ni byly obsazeny pouze hrany sousednich
    elementu urcene k sekundarnimu presitovani
354     cnt = 0
355     while np.any(Hrana[:,-1] == 0):
356         if cnt >= Hrana.shape[0]:
357             break
358         elif Hrana[cnt,-1] == 0:
359             Hrana = np.delete(Hrana,cnt, axis=0)
360         else:
361             cnt += 1
362
363 #Serazeni matice Hrana dle indexu elementu - vzestupne
364     col = 2
365     Hrana = Hrana[np.argsort(Hrana[:,col])]
366     Elementy_dositovani = Hrana[:,2]
367     Pojistka = -1
368
369     for j in range(Elementy_dositovani.shape[0]):
370         Vyskyt = np.count_nonzero(Hrana[:,2] == Elementy_dositovani[j])
371 #Presitovani ctyr-uzlovych sousedu rozbitim na 3 trojuhelniky
372 #         if Vyskyt == 1:
373 #             Pripad = np.zeros((3), dtype=np.int32)
374 #             cnt = 0
```

```
375 #             for k in range(4):
376 #
377 #                 if np.any(elements[Elementy_dositovani[j],k] == Hrana[j
    ,:2]):
378 #                     continue
379 #                 else:
380 #                     if k == 3:
381 #                         Pripad[-1] += (k+1)
382 #                         Pripad[cnt] = elements[Elementy_dositovani[j],k]
383 #                         cnt += 1
384 #                     else:
385 #                         Pripad[-1] += k
386 #                         Pripad[cnt] = elements[Elementy_dositovani[j],k]
387 #                         cnt += 1
388 #
389 #             if Pripad[-1] == 1:
390 #                 Node_1 = elements[Elementy_dositovani[j],-1]
391 #                 Node_2 = elements[Elementy_dositovani[j],2]
392 #
393 #             elif Pripad[-1] == 3:
394 #                 Node_1 = elements[Elementy_dositovani[j],0]
395 #                 Node_2 = elements[Elementy_dositovani[j],-1]
396 #
397 #             elif Pripad[-1] == 6:
398 #                 Node_1 = elements[Elementy_dositovani[j],1]
399 #                 Node_2 = elements[Elementy_dositovani[j],0]
400 #
401 #             elif Pripad[-1] == 4:
402 #                 Pripad[:] = [Pripad[1],Pripad[0],Pripad[-1]]
403 #                 Node_1 = elements[Elementy_dositovani[j],2]
404 #                 Node_2 = elements[Elementy_dositovani[j],1]
405 #
406 #
407 #             elements[Elementy_dositovani[j],:] = [Pripad[0],Pripad[1],
    Hrana[j,-1],-1]
408 #             elements = np.append(elements,[[Pripad[0],Hrana[j,-1],Node_1
    ,-1]],0)
409 #             elements = np.append(elements,[[Pripad[1],Node_2,Hrana[j
    ,-1],-1]],0)
410
```

```
411 #Presitovani ctyr-uzlovych sousedu rozbitim na vice nez 3 trojuhelniky
412         if Vyskyt >= 1:
413 #Vypocet souradnic teziste
414             if Elementy_dositovani[j] == Pojistka:
415                 pass
416 
417             else:
418                 Old_one = elements[Elementy_dositovani[j],:]
419                 Area_C = 0.0
420                 Vaha_x = 0.0
421                 Vaha_y = 0.0
422                 for i in range(2):
423                     ie = elements[Elementy_dositovani[j],:]
424                     ie_t = [ie[0],ie[i+1],ie[i+2]]
425                     xi = coordinates[ie_t,0]
426                     yi = coordinates[ie_t,1]
427                     Ah_ = np.array([ones,xi,yi]).transpose()
428                     Area_h = np.linalg.det(Ah_)*0.5
429                     x_tt = (1/3)*(xi[0] + xi[1] + xi[2])
430                     y_tt = (1/3)*(yi[0] + yi[1] + yi[2])
431 
432                     Vaha_x += Area_h*x_tt
433                     Vaha_y += Area_h*y_tt
434                     Area_C += Area_h
435 
436                     x_T = Vaha_x/Area_C
437                     y_T = Vaha_y/Area_C
438 
439                 coordinates = np.append(coordinates,[[x_T,y_T]],0)
440                 Teziste = coordinates.shape[0]-1
441 
442 
443                 Position = np.zeros((4),dtype=np.int32)
444                 for i in range(3):
445                     elements = np.append(elements,[[elements[
    Elementy_dositovani[j],i],elements[Elementy_dositovani[j],1+i],Teziste
    ,-1]],0)
446                     Position[i] = elements.shape[0]-1
447                 elements[Elementy_dositovani[j],:] = [elements[
    Elementy_dositovani[j],3],elements[Elementy_dositovani[j],0],Teziste
```

```
,-1]
448                 Position[-1] = Elementy_dositovani[j]
449
450             if Hrana[j,3] <= 1:
451                 if Hrana[j,3] == 0:
452                     elements[Position[0],:] = [Old_one[0],Hrana[j,4],
    Teziste,-1]
453                     elements = np.append(elements,[[Hrana[j,4],Old_one
    [1],Teziste,-1]],0)
454
455                 if Hrana[j,3] == 1:
456                     elements[Position[1],:] = [Old_one[1],Hrana[j,4],
    Teziste,-1]
457                     elements = np.append(elements,[[Hrana[j,4],Old_one
    [2],Teziste,-1]],0)
458
459             else:
460                 if Hrana[j,3] == 2:
461                     elements[Position[2],:] = [Old_one[2],Hrana[j,4],
    Teziste,-1]
462                     elements = np.append(elements,[[Hrana[j,4],Old_one
    [3],Teziste,-1]],0)
463
464                 if Hrana[j,3] == 3:
465                     elements[Position[3],:] = [Old_one[3],Hrana[j,4],
    Teziste,-1]
466                     elements = np.append(elements,[[Hrana[j,4],Old_one
    [0],Teziste,-1]],0)
467
468             Pojistka =    Elementy_dositovani[j]
469
470     if Adaptace_na_trojuhelniky.shape[0] > 0:
471         for j in range(Adaptace_na_trojuhelniky.shape[0]):
472             Area_C = 0.0
473             Vaha_x = 0.0
474             Vaha_y = 0.0
475             for i in range(2):
476                 ie = elements[Adaptace_na_trojuhelniky[j],:]
477                 ie_t = [ie[0],ie[i+1],ie[i+2]]
478                 xi = coordinates[ie_t,0]
```

```
479                 yi = coordinates[ie_t,1]
480                 Ah_ = np.array([ones,xi,yi]).transpose()
481                 Area_h = np.linalg.det(Ah_)*0.5
482                 x_tt = (1/3)*(xi[0] + xi[1] + xi[2])
483                 y_tt = (1/3)*(yi[0] + yi[1] + yi[2])
484
485                 Vaha_x += Area_h*x_tt
486                 Vaha_y += Area_h*y_tt
487                 Area_C += Area_h
488
489             x_T = Vaha_x/Area_C
490             y_T = Vaha_y/Area_C
491
492             coordinates = np.append(coordinates,[[x_T,y_T]],0)
493             Teziste = coordinates.shape[0]-1
494
495             for i in range(3):
496                 elements = np.append(elements,[[elements[
    Adaptace_na_trojuhelniky[j],i],elements[Adaptace_na_trojuhelniky[j],1+
    i],Teziste,-1]],0)
497
498             elements[Adaptace_na_trojuhelniky[j],:] = [elements[
    Adaptace_na_trojuhelniky[j],3],elements[Adaptace_na_trojuhelniky[j
    ],0],Teziste,-1]
499
500     return coordinates,elements,edge0,edge1,edge2,edge3
```

Příloha B - Standardní strategie dělení trojúhelníkových elementů

```
import numpy as np

def Remesh(coordinates,Rem_elements,elements,edge0,edge1,edge2,edge3,R1,
    R0,Pos,Hrana,Tl_nadoba):
     for j in range(Rem_elements.shape[0]):
          if Rem_elements[j,-1] == -1:
#=============================================================================
#Zjisteni pozic hran vybraneho elementu v matici Hrana + serazeni dle
    cisla hrany
               Pozice = np.zeros((3,2),dtype=np.int32)
               cnt = 0
               for i, k in enumerate(Hrana):
                    if Hrana[i,2] == Pos[j]:
                         Pozice[cnt,0] = i
                         Pozice[cnt,1] = Hrana[i,3]
                         cnt += 1

               col = 1
               Pozice = Pozice[np.argsort(Pozice[:,col])]
               Pozice = np.delete(Pozice,1,1)
#=============================================================================
#Detekce sousednich elementu
               Soused = np.zeros((3,5),dtype=np.int32)
               for i in range(3):
                    if Pozice[i,0] == 0:
                         cnt = 1
                         while Hrana[Pozice[i,0],0] == Hrana[Pozice[i,0]+cnt
    ,0]:
                              if np.all(Hrana[Pozice[i,0],:2] == Hrana[Pozice[i
    ,0]+cnt,:2]):
                                   Soused[i,:3] = Hrana[Pozice[i,0]+cnt,:3]
                                   Soused[i,3]  = Pozice[i,0]+cnt
                                   Soused[i,-1] = i
                                   break
                              else:
                                   cnt += 1

```

```
34                  if Pozice[i,0] == Hrana.shape[0]-1:
35                      cnt = 1
36                      while Hrana[Pozice[i,0],0] == Hrana[Pozice[i,0]-cnt
    ,0]:
37                          if np.all(Hrana[Pozice[i,0],:2] == Hrana[Pozice[i
    ,0]-cnt,:2]):
38                              Soused[i,:3] = Hrana[Pozice[i,0]-cnt,:3]
39                              Soused[i,3]  = Pozice[i,0]-cnt
40                              Soused[i,-1] = i
41                              break
42                          else:
43                              cnt += 1
44
45                  if 0 < Pozice[i,0] < Hrana.shape[0]-1:
46                      for k in range(2):
47                          if k == 0:
48                              cnt = 1
49
50                              while Hrana[Pozice[i,0],0] == Hrana[Pozice[i
    ,0]-cnt,0]:
51                                  if np.all(Hrana[Pozice[i,0],:2] == Hrana[
    Pozice[i,0]-cnt,:2]):
52                                      Soused[i,:3] = Hrana[Pozice[i,0]-cnt
    ,:3]
53                                      Soused[i,3]  = Pozice[i,0]-cnt
54                                      Soused[i,-1] = i
55                                      break
56                                  else:
57                                      cnt += 1
58
59                          if k == 1:
60                              cnt = 1
61                              while Hrana[Pozice[i,0],0] == Hrana[Pozice[i
    ,0]+cnt,0]:
62                                  if np.all(Hrana[Pozice[i,0],:2] == Hrana[
    Pozice[i,0]+cnt,:2]):
63                                      Soused[i,:3] = Hrana[Pozice[i,0]+cnt
    ,:3]
64                                      Soused[i,3]  = Pozice[i,0]+cnt
65                                      Soused[i,-1] = i
```

```
66                                      break
67                                  elif Pozice[i,0] + cnt == Hrana.shape
    [0]-1:
68                                      break
69                                  else:
70                                      cnt += 1
71
72 #Pokud hrana elementu nema souseda, je stale zapotrebi oznacit cislo
    hrany kvuli naslednemu serazeni
73                 if np.all(Soused[i,:] == 0):
74                     Soused[i,-1] = i
75 #Serazeni sousedu
76             col = 4
77             Soused = Soused[np.argsort(Soused[:,col])]
78
79 #Pokud doslo k pripadu zminenem vyse, je zapotrebi po serazeni cislo
    hrany prepsat nulou pro spravnou funkci algoritmu
80             for i in range(3):
81                 if np.all(Soused[i,:4] == 0):
82                     Soused[i,-1] = 0
83
84 #=============================================================================
85 #Stanoveni verze presitovani
86             Typ = False
87             if Rem_elements.shape[0] == elements.shape[0]:
88                 Typ = True
89
90 #=============================================================================
91
92 #Pokud je Typ == True, provede se deleni na 4 trojuhelniky
93             if Typ == True:
94                 T = np.zeros((3),dtype=np.int32)
95                 for i in range(3):
96                     x_T = 0.5*(coordinates[Hrana[Pozice[i,0],0],0]+
    coordinates[Hrana[Pozice[i,0],1],0])
97                     y_T = 0.5*(coordinates[Hrana[Pozice[i,0],0],1]+
    coordinates[Hrana[Pozice[i,0],1],1])
98
99                     for k in range(coordinates.shape[0]):
100                        if coordinates[k,0] == x_T:
```

```
101                             if coordinates[k,1] == y_T:
102                                 T[i] = k
103                                 break
104 
105                             elif k == coordinates.shape[0]-1:
106                                 coordinates = np.append(coordinates,[[x_T
    ,y_T]],0)
107                                 T[i] = coordinates.shape[0]-1
108                                 break
109 
110                         elif k == coordinates.shape[0]-1:
111                             coordinates = np.append(coordinates,[[x_T,y_T
    ]],0)
112                             T[i] = coordinates.shape[0]-1
113 
114                     if np.all(Soused[i,:] == 0):
115                         continue
116                     else:
117                         Hrana[Soused[i,3],-1] = T[i]
118                 elements[Pos[j],:] = [[Rem_elements[j,0],T[0],T[-1],-1]]
119                 elements = np.append(elements,[[T[0],Rem_elements[j,1],T
    [1],-1]],0)
120                 elements = np.append(elements,[[T[1],Rem_elements[j,2],T
    [2],-1]],0)
121                 elements = np.append(elements,[[T[0],T[1],T[2],-1]],0)
122 
123 
124 #=================================================================
125 #Aktualizace edge0-4
126                 for i in range(edge0.shape[0]):
127                     for k in range(2):
128                         Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
129                         if np.any(edge0[i,:] == Vybr_par[0]):
130                             if np.any(edge0[i,:] == Vybr_par[1]):
131                                 edge0[i,0] = Vybr_par[0]
132                                 edge0[i,1] = T[k]
133                                 edge0 = np.append(edge0,[[Vybr_par[1],T[k
    ]]],0)
134 
```

```
135 #Presun noveho uzlu na vnitrni polomer nadoby
136                                 if Tl_nadoba == True:
137                                     p = np.array([coordinates[T[k],0],
    coordinates[T[k],1]])
138                                     c = R0/(p[0]**2+p[1]**2)**0.5
139                                     coordinates[T[k],0] = p[0]*c
140                                     coordinates[T[k],1] = p[1]*c
141
142                     Vybr_par = np.array([Rem_elements[j,2],Rem_elements[j
    ,0]])
143                     if np.any(edge0[i,:] == Vybr_par[0]):
144                         if np.any(edge0[i,:] == Vybr_par[1]):
145                             edge0[i,0] = Vybr_par[0]
146                             edge0[i,1] = T[2]
147                             edge0 = np.append(edge0,[[Vybr_par[1],T
    [2]]],0)
148
149 #Presun noveho uzlu na vnitrni polomer nadoby
150                                 if Tl_nadoba == True:
151                                     p = np.array([coordinates[T[2],0],
    coordinates[T[2],1]])
152                                     c = R0/(p[0]**2+p[1]**2)**0.5
153                                     coordinates[T[2],0] = p[0]*c
154                                     coordinates[T[2],1] = p[1]*c
155
156                 for i in range(edge1.shape[0]):
157                     for k in range(2):
158                         Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
159                         if np.any(edge1[i,:] == Vybr_par[0]):
160                             if np.any(edge1[i,:] == Vybr_par[1]):
161                                 edge1[i,0] = Vybr_par[0]
162                                 edge1[i,1] = T[k]
163                                 edge1 = np.append(edge1,[[Vybr_par[1],T[k
    ]]],0)
164
165                         Vybr_par = np.array([Rem_elements[j,2],
    Rem_elements[j,0]])
166                         if np.any(edge1[i,:] == Vybr_par[0]):
167                             if np.any(edge1[i,:] == Vybr_par[1]):
```

```
168                                 edge1[i,0] = Vybr_par[0]
169                                 edge1[i,1] = T[2]
170                                 edge1 = np.append(edge1,[[Vybr_par[1],T
    [2]]],0)
171
172
173                     for i in range(edge2.shape[0]):
174                         for k in range(2):
175                             Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
176                             if np.any(edge2[i,:] == Vybr_par[0]):
177                                 if np.any(edge2[i,:] == Vybr_par[1]):
178                                     edge2[i,0] = Vybr_par[0]
179                                     edge2[i,1] = T[k]
180                                     edge2 = np.append(edge2,[[Vybr_par
    [1],T[k]]],0)
181
182 #Presun noveho uzlu na vnejsi polomer nadoby
183                                     if Tl_nadoba == True:
184                                         p = np.array([coordinates[T[k
    ],0],coordinates[T[k],1]])
185                                         c = R1/(p[0]**2+p[1]**2)**0.5
186                                         coordinates[T[k],0] = p[0]*c
187                                         coordinates[T[k],1] = p[1]*c
188
189                         Vybr_par = np.array([Rem_elements[j,2],
    Rem_elements[j,0]])
190                         if np.any(edge2[i,:] == Vybr_par[0]):
191                             if np.any(edge2[i,:] == Vybr_par[1]):
192                                 edge2[i,0] = Vybr_par[0]
193                                 edge2[i,1] = T[2]
194                                 edge2 = np.append(edge2,[[Vybr_par[1],T
    [2]]],0)
195
196 #Presun noveho uzlu na vnejsi polomer nadoby
197                                 if Tl_nadoba == True:
198                                     p = np.array([coordinates[T[2],0],
    coordinates[T[2],1]])
199                                     c = R1/(p[0]**2+p[1]**2)**0.5
200                                     coordinates[T[2],0] = p[0]*c
```

```
                                    coordinates[T[2],1] = p[1]*c


                    for i in range(edge3.shape[0]):
                        for k in range(2):
                            Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
                            if np.any(edge3[i,:] == Vybr_par[0]):
                                if np.any(edge3[i,:] == Vybr_par[1]):
                                    edge3[i,0] = Vybr_par[0]
                                    edge3[i,1] = T[k]
                                    edge3 = np.append(edge3,[[Vybr_par
    [1],T[k]]],0)

                        Vybr_par = np.array([Rem_elements[j,2],
    Rem_elements[j,0]])
                        if np.any(edge3[i,:] == Vybr_par[0]):
                            if np.any(edge3[i,:] == Vybr_par[1]):
                                edge3[i,0] = Vybr_par[0]
                                edge3[i,1] = T[2]
                                edge3 = np.append(edge3,[[Vybr_par[1],T
    [2]]],0)


#=====================================================================


            if Typ == False:

                u = np.zeros(3)
                for i in range(3):
                    if i < 2:
                        u[i] = ((coordinates[Rem_elements[j,i],0]-
    coordinates[Rem_elements[j,i+1],0])**2 +\
                                (coordinates[Rem_elements[j,i],1]-
    coordinates[Rem_elements[j,i+1],1])**2)**0.5

                    else:
                        u[i] = ((coordinates[Rem_elements[j,0],0]-
    coordinates[Rem_elements[j,2],0])**2 +\
```

```
234                                 (coordinates[Rem_elements[j,0],1]-
    coordinates[Rem_elements[j,2],1])**2)**0.5
235
236 #=====================================================================
237 #Stanoveni pozice maximalni hodnoty z vektoru velikosti stran
    trojuhelnikoveho elementu
238                 Place = np.argmax(u)
239 #=====================================================================
240 #Vypocet souradnice noveho uzlu ve stredu nejvetsi strany
241                 for k in range(1):
242                     if Place == 0:
243                         x_T = 0.5*(coordinates[Rem_elements[j,0],0]+
    coordinates[Rem_elements[j,1],0])
244                         y_T = 0.5*(coordinates[Rem_elements[j,0],1]+
    coordinates[Rem_elements[j,1],1])
245                         Vybr_par = np.array([Rem_elements[j,0],
    Rem_elements[j,1]])
246
247                     elif Place == 1:
248                         x_T = 0.5*(coordinates[Rem_elements[j,1],0]+
    coordinates[Rem_elements[j,2],0])
249                         y_T = 0.5*(coordinates[Rem_elements[j,1],1]+
    coordinates[Rem_elements[j,2],1])
250                         Vybr_par = np.array([Rem_elements[j,1],
    Rem_elements[j,2]])
251
252                     elif Place == 2:
253                         x_T = 0.5*(coordinates[Rem_elements[j,0],0]+
    coordinates[Rem_elements[j,2],0])
254                         y_T = 0.5*(coordinates[Rem_elements[j,0],1]+
    coordinates[Rem_elements[j,2],1])
255                         Vybr_par = np.array([Rem_elements[j,0],
    Rem_elements[j,2]])
256
257                 T = np.zeros((1),dtype=np.int32)
258                 for k in range(coordinates.shape[0]):
259                     if coordinates[k,0] == x_T:
260                         if coordinates[k,1] == y_T:
261                             T[0] = k
262                             break
```

```
263
264                         elif k == coordinates.shape[0]-1:
265                             coordinates = np.append(coordinates,[[x_T,y_T
    ]],0)
266                             T[0] = coordinates.shape[0]-1
267                             break
268
269                     elif k == coordinates.shape[0]-1:
270                         coordinates = np.append(coordinates,[[x_T,y_T
    ]],0)
271                         T[0] = coordinates.shape[0]-1
272
273                 for k in range(1):
274                     if np.all(Soused[Place,:] == 0):
275                         continue
276                     else:
277                         Hrana[Soused[Place,3],-1] = T[0]
278
279                 for k in range(1):
280                     if Place == 0:
281                         elements[Pos[j],:] = [[T[0],Rem_elements[j,1],
    Rem_elements[j,2],-1]]
282                         elements = np.append(elements,[[T[0],Rem_elements
    [j,2],Rem_elements[j,0],-1]],0)
283                         Hrana[Pozice[2],2] = elements.shape[0]-1
284                         Hrana[Pozice[1],3] = 1
285                         Hrana[Pozice[2],3] = 1
286
287                     elif Place == 1:
288                         elements[Pos[j],:] = [[T[0],Rem_elements[j,2],
    Rem_elements[j,0],-1]]
289                         elements = np.append(elements,[[T[0],Rem_elements
    [j,0],Rem_elements[j,1],-1]],0)
290                         Hrana[Pozice[0],2] = elements.shape[0]-1
291                         Hrana[Pozice[0],3] = 1
292                         Hrana[Pozice[2],3] = 1
293
294                     elif Place == 2:
295                         elements[Pos[j],:] = [[T[0],Rem_elements[j,0],
    Rem_elements[j,1],-1]]
```

```
296                         elements = np.append(elements,[[T[0],Rem_elements
    [j,1],Rem_elements[j,2],-1]],0)
297                         Hrana[Pozice[1],2] = elements.shape[0]-1
298                         Hrana[Pozice[1],3] = 1
299                         Hrana[Pozice[0],3] = 1
300
301 #=====================================================================
302 #Aktualizace edge0-4
303                 for i in range(edge0.shape[0]):
304                     for k in range(2):
305                         if np.any(edge0[i,:] == Vybr_par[0]):
306                             if np.any(edge0[i,:] == Vybr_par[1]):
307                                 edge0[i,0] = Vybr_par[0]
308                                 edge0[i,1] = T[0]
309                                 edge0 = np.append(edge0,[[Vybr_par[1],T
    [0]]],0)
310
311 #Presun noveho uzlu na vnitrni polomer nadoby
312                                 if Tl_nadoba == True:
313                                     p = np.array([coordinates[T[0],0],
    coordinates[T[0],1]])
314                                     c = R0/(p[0]**2+p[1]**2)**0.5
315                                     coordinates[T[0],0] = p[0]*c
316                                     coordinates[T[0],1] = p[1]*c
317
318                 for i in range(edge1.shape[0]):
319                     for k in range(2):
320                         if np.any(edge1[i,:] == Vybr_par[0]):
321                             if np.any(edge1[i,:] == Vybr_par[1]):
322                                 edge1[i,0] = Vybr_par[0]
323                                 edge1[i,1] = T[0]
324                                 edge1 = np.append(edge1,[[Vybr_par[1],T
    [0]]],0)
325
326                 for i in range(edge2.shape[0]):
327                     for k in range(2):
328                         if np.any(edge2[i,:] == Vybr_par[0]):
329                             if np.any(edge2[i,:] == Vybr_par[1]):
330                                 edge2[i,0] = Vybr_par[0]
331                                 edge2[i,1] = T[0]
```

```
332                                 edge2 = np.append(edge2,[[Vybr_par[1],T
    [0]]],0)
333
334 #Presun noveho uzlu na vnejsi polomer nadoby
335                                 if Tl_nadoba == True:
336                                     p = np.array([coordinates[T[0],0],
    coordinates[T[0],1]])
337                                     c = R1/(p[0]**2+p[1]**2)**0.5
338                                     coordinates[T[0],0] = p[0]*c
339                                     coordinates[T[0],1] = p[1]*c
340
341
342                 for i in range(edge3.shape[0]):
343                     for k in range(2):
344                         if np.any(edge3[i,:] == Vybr_par[0]):
345                             if np.any(edge3[i,:] == Vybr_par[1]):
346                                 edge3[i,0] = Vybr_par[0]
347                                 edge3[i,1] = T[0]
348                                 edge3 = np.append(edge3,[[Vybr_par[1],T
    [0]]],0)
349
350
351 #=========================================================================
352 #Dositovani sousednich elementu, ktere nebyly urceny k presitovani, ale
    na jejich hranach vznikli nove uzly
353
354 #Uprava matice Hrana tak, aby v ni byly obsazeny pouze hrany sousednich
    elementu urcene k dositovani
355     cnt = 0
356     while np.any(Hrana[:,-1] == 0):
357         if cnt >= Hrana.shape[0]:
358             break
359         elif Hrana[cnt,-1] == 0:
360             Hrana = np.delete(Hrana,cnt, axis=0)
361         else:
362             cnt += 1
363
364     cnt = 0
365     Hrana = Hrana[np.argsort(Hrana[:,4])]
366     while np.any(Hrana[:,-1] == Hrana[:,-1]):
```

```
367             if cnt+1 >= Hrana.shape[0]:
368                 break
369             elif Hrana[cnt,-1] == Hrana[cnt+1,-1]:
370                 Hrana = np.delete(Hrana,cnt, axis=0)
371                 Hrana = np.delete(Hrana,cnt, axis=0)
372             else:
373                 cnt += 1
374 
375     Hrana = Hrana[np.argsort(Hrana[:,2])]
376     Elementy_dositovani = Hrana[:,2]
377     Pojistka = -1
378     ones = np.ones(3)
379     for j in range(Elementy_dositovani.shape[0]):
380         Vyskyt = np.count_nonzero(Hrana[:,2] == Elementy_dositovani[j])
381         if Vyskyt == 1:
382             if elements[Hrana[j,2],-1] == -1:
383                 pass
384                 if Hrana[j,3] == 0:
385                     elements = np.append(elements,[[Hrana[j,4],elements[
    Hrana[j,2],1],elements[Hrana[j,2],2],-1]],0)
386                     elements[Hrana[j,2],:] = [Hrana[j,4],elements[Hrana[j
    ,2],2],elements[Hrana[j,2],0],-1]
387 
388                 if Hrana[j,3] == 1:
389                     elements = np.append(elements,[[Hrana[j,4],elements[
    Hrana[j,2],2],elements[Hrana[j,2],0],-1]],0)
390                     elements[Hrana[j,2],:] = [Hrana[j,4],elements[Hrana[j
    ,2],0],elements[Hrana[j,2],1],-1]
391 
392                 elif Hrana[j,3] == 2:
393                     elements = np.append(elements,[[Hrana[j,4],elements[
    Hrana[j,2],0],elements[Hrana[j,2],1],-1]],0)
394                     elements[Hrana[j,2],:] = [Hrana[j,4],elements[Hrana[j
    ,2],1],elements[Hrana[j,2],2],-1]
395 
396             else:
397 #Presitovani ctyr-uzlovych sousedu rozbitim na vice nez 3 trojuhelniky
398     #Vypocet souradnic teziste
399                 if Elementy_dositovani[j] == Pojistka:
400                     pass
```

```
401                 else:
402                     Old_one = elements[Elementy_dositovani[j],:]
403                     Area_C = 0.0
404                     Vaha_x = 0.0
405                     Vaha_y = 0.0
406                     for i in range(2):
407                         ie = elements[Elementy_dositovani[j],:]
408                         ie_t = [ie[0],ie[i+1],ie[i+2]]
409                         xi = coordinates[ie_t,0]
410                         yi = coordinates[ie_t,1]
411                         Ah_ = np.array([ones,xi,yi]).transpose()
412                         Area_h = np.linalg.det(Ah_)*0.5
413                         x_tt = (1/3)*(xi[0] + xi[1] + xi[2])
414                         y_tt = (1/3)*(yi[0] + yi[1] + yi[2])
415
416                         Vaha_x += Area_h*x_tt
417                         Vaha_y += Area_h*y_tt
418                         Area_C += Area_h
419
420                         x_T = Vaha_x/Area_C
421                         y_T = Vaha_y/Area_C
422
423                     coordinates = np.append(coordinates,[[x_T,y_T]],0)
424                     Teziste = coordinates.shape[0]-1
425
426
427                     Position = np.zeros((4),dtype=np.int32)
428                     for i in range(3):
429                         elements = np.append(elements,[[elements[
    Elementy_dositovani[j],i],elements[Elementy_dositovani[j],1+i],Teziste
    ,-1]],0)
430                         Position[i] = elements.shape[0]-1
431                     elements[Elementy_dositovani[j],:] = [elements[
    Elementy_dositovani[j],3],elements[Elementy_dositovani[j],0],Teziste
    ,-1]
432                     Position[-1] = Elementy_dositovani[j]
433
434                 if Hrana[j,3] <= 1:
435                     if Hrana[j,3] == 0:
436                         elements[Position[0],:] = [Old_one[0],Hrana[j,4],
```

```
    Teziste,-1]
437                         elements = np.append(elements,[[Hrana[j,4],
    Old_one[1],Teziste,-1]],0)
438
439                     if Hrana[j,3] == 1:
440                         elements[Position[1],:] = [Old_one[1],Hrana[j,4],
    Teziste,-1]
441                         elements = np.append(elements,[[Hrana[j,4],
    Old_one[2],Teziste,-1]],0)
442
443                 else:
444                     if Hrana[j,3] == 2:
445                         elements[Position[2],:] = [Old_one[2],Hrana[j,4],
    Teziste,-1]
446                         elements = np.append(elements,[[Hrana[j,4],
    Old_one[3],Teziste,-1]],0)
447
448                     if Hrana[j,3] == 3:
449                         elements[Position[3],:] = [Old_one[3],Hrana[j,4],
    Teziste,-1]
450                         elements = np.append(elements,[[Hrana[j,4],
    Old_one[0],Teziste,-1]],0)
451
452                 Pojistka =    Elementy_dositovani[j]
453
454     return coordinates,elements,edge0,edge1,edge2,edge3
```

Příloha C - Modifikovaná strategie dělení čtyřúhelníkových elementů

```
import numpy as np
from datetime import datetime

def Pure(coordinates,Rem_elements,elements,Pos,tvar,edge0,edge1,edge2,
    edge3,R1,R0,Hrana,Hrana_memory,Tl_nadoba):
    ones = np.ones(3)
#=====================================================================
#Urceni teziste ctyruzloveho prvku
    for j in range(Rem_elements.shape[0]):
        if Rem_elements[j,-1] >= 0:
                Area_C = 0.0
                Vaha_x = 0.0
                Vaha_y = 0.0
                for i in range(2):
                    ie = Rem_elements[j,:]
                    ie_t = [ie[0],ie[i+1],ie[i+2]]
                    xi = coordinates[ie_t,0]
                    yi = coordinates[ie_t,1]
                    Ah_ = np.array([ones,xi,yi]).transpose()
                    Area_h = np.linalg.det(Ah_)*0.5
                    x_tt = (1/3)*(xi[0] + xi[1] + xi[2])
                    y_tt = (1/3)*(yi[0] + yi[1] + yi[2])

                    Vaha_x += Area_h*x_tt
                    Vaha_y += Area_h*y_tt
                    Area_C += Area_h

                    x_T = Vaha_x/Area_C
                    y_T = Vaha_y/Area_C

                coordinates = np.append(coordinates,[[x_T,y_T]],0)
                Teziste = coordinates.shape[0]-1

#=====================================================================
#Zjisteni pozic hran vybraneho elementu v matici Hrana + serazeni dle
    cisla hrany
                Pozice = np.zeros((4,2),dtype=np.int32)
```

```
36                  cnt = 0
37                  for i, k in enumerate(Hrana):
38                      if Hrana[i,2] == Pos[j]:
39                          Pozice[cnt,0] = i
40                          Pozice[cnt,1] = Hrana[i,3]
41                          cnt += 1
42                  col = 1
43                  Pozice = Pozice[np.argsort(Pozice[:,col])]
44                  Pozice = np.delete(Pozice,1,1)
45 #=====================================================================
46 #Detekce sousednich elementu
47
48                  Soused = np.zeros((4,5),dtype=np.int32)
49                  for i in range(4):
50                      if Pozice[i,0] == 0:
51                          cnt = 1
52                          while Hrana[Pozice[i,0],0] == Hrana[Pozice[i,0]+
    cnt,0]:
53                              if np.all(Hrana[Pozice[i,0],:2] == Hrana[
    Pozice[i,0]+cnt,:2]):
54                                  Soused[i,:3] = Hrana[Pozice[i,0]+cnt,:3]
55                                  Soused[i,3]  = Pozice[i,0]+cnt
56                                  Soused[i,-1] = i
57                                  break
58                              else:
59                                  cnt += 1
60
61                      if Pozice[i,0] == Hrana.shape[0]-1:
62                          cnt = 1
63                          while Hrana[Pozice[i,0],0] == Hrana[Pozice[i,0]-
    cnt,0]:
64                              if np.all(Hrana[Pozice[i,0],:2] == Hrana[
    Pozice[i,0]-cnt,:2]):
65                                  Soused[i,:3] = Hrana[Pozice[i,0]-cnt,:3]
66                                  Soused[i,3]  = Pozice[i,0]-cnt
67                                  Soused[i,-1] = i
68                                  break
69                              else:
70                                  cnt += 1
71
```

```
72                      if 0 < Pozice[i,0] < Hrana.shape[0]-1:
73                          for k in range(2):
74                              if k == 0:
75                                  cnt = 1
76
77                                  while Hrana[Pozice[i,0],0] == Hrana[
    Pozice[i,0]-cnt,0]:
78                                      if np.all(Hrana[Pozice[i,0],:2] ==
    Hrana[Pozice[i,0]-cnt,:2]):
79                                          Soused[i,:3] = Hrana[Pozice[i,0]-
    cnt,:3]
80                                          Soused[i,3]  = Pozice[i,0]-cnt
81                                          Soused[i,-1] = i
82                                          break
83                                      else:
84                                          cnt += 1
85
86                              if k == 1:
87                                  cnt = 1
88                                  while Hrana[Pozice[i,0],0] == Hrana[
    Pozice[i,0]+cnt,0]:
89                                      if np.all(Hrana[Pozice[i,0],:2] ==
    Hrana[Pozice[i,0]+cnt,:2]):
90                                          Soused[i,:3] = Hrana[Pozice[i,0]+
    cnt,:3]
91                                          Soused[i,3]  = Pozice[i,0]+cnt
92                                          Soused[i,-1] = i
93                                          break
94                                      elif Pozice[i,0] + cnt == Hrana.shape
    [0]-1:
95                                          break
96                                      else:
97                                          cnt += 1
98
99  #Pokud hrana elementu nema souseda, je stale zapotrebi oznacit cislo
    hrany kvuli naslednemu serazeni
100                     if np.all(Soused[i,:] == 0):
101                         Soused[i,-1] = i
102
103 #Serazeni sousedu
```

```
104                 col = 4
105                 Soused = Soused[np.argsort(Soused[:,col])]
106 
107 #Pokud doslo k pripadu zminenem vyse, je zapotrebi po serazeni cislo hrany prepsat nulou pro spravnou funkci algoritmu
108                 for i in range(4):
109                     if np.all(Soused[i,:4] == 0):
110                         Soused[i,-1] = 0
111 #=====================================================================
112 #Kontrola zda sousedni element neni trojuhelnik a zda se bude sousedni element presitovavat
113                 for i in range(4):
114                     if np.all(Soused[i,:] == 0):
115                         continue
116                     else:
117 #Pokud je sousedni element urcen k presitovani, neni potreba se s nim zaobirat
118                         if np.any(Pos[:] == Soused[i,2]):
119                             Soused[i,:] = 0
120                         else:
121                             continue
122 #=====================================================================
123 #Pokud je ctyruzlovy prvek zdeformovany rozdeli se na 4 trojuhelniky
124                 if tvar[j] == 1:
125                     for i in range(3):
126                         elements = np.append(elements,[[Rem_elements[j,i],Rem_elements[j,1+i],Teziste,-1]],0)
127 
128                     elements[Pos[j],:] = [[Rem_elements[j,3],Rem_elements[j,0],Teziste,-1]]
129 
130 #=====================================================================
131 #Pokud je ctyruzlovy prvek nezdeformovany, lze rozdelit na 4 mensi ctyruhelniky
132                 else:
133                     T = np.zeros((4),dtype=np.int32)
134                     for i in range(4):
135                         x_T = 0.5*(coordinates[Hrana[Pozice[i,0],0],0]+coordinates[Hrana[Pozice[i,0],1],0])
136                         y_T = 0.5*(coordinates[Hrana[Pozice[i,0],0],1]+
```

```
    coordinates[Hrana[Pozice[i,0],1],1])
137
138                         for k in range(coordinates.shape[0]):
139                             if coordinates[k,0] == x_T:
140                                 if coordinates[k,1] == y_T:
141                                     T[i] = k
142                                     break
143
144                                 elif k == coordinates.shape[0]-1:
145                                     coordinates = np.append(coordinates
    ,[[x_T,y_T]],0)
146                                     T[i] = coordinates.shape[0]-1
147                                     break
148
149                             elif k == coordinates.shape[0]-1:
150                                 coordinates = np.append(coordinates,[[x_T
    ,y_T]],0)
151                                 T[i] = coordinates.shape[0]-1
152
153                         if np.all(Soused[i,:] == 0):
154                             continue
155                         else:
156                             Hrana[Soused[i,3],-1] = T[i]
157
158                     elements[Pos[j],:] = [[Rem_elements[j,0],T[0],Teziste
    ,T[3]]]
159                     elements = np.append(elements,[[T[0],Rem_elements[j
    ,1],T[1],Teziste]],0)
160                     elements = np.append(elements,[[Teziste,T[1],
    Rem_elements[j,2],T[2]]],0)
161                     elements = np.append(elements,[[T[3],Teziste,T[2],
    Rem_elements[j,3]]],0)
162
163 #=====================================================================
164 #Aktualizace edge0-4
165                     for i in range(edge0.shape[0]):
166                         for k in range(3):
167                             Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
168                             if np.any(edge0[i,:] == Vybr_par[0]):
```

```
169                                 if np.any(edge0[i,:] == Vybr_par[1]):
170                                     edge0[i,0] = Vybr_par[0]
171                                     edge0[i,1] = T[k]
172                                     edge0 = np.append(edge0,[[Vybr_par
    [1],T[k]]],0)
173
174 #Presun noveho uzlu na vnitrni polomer nadoby
175                                     if Tl_nadoba == True:
176                                         p = np.array([coordinates[T[k
    ],0],coordinates[T[k],1]])
177                                         c = R0/(p[0]**2+p[1]**2)**0.5
178                                         coordinates[T[k],0] = p[0]*c
179                                         coordinates[T[k],1] = p[1]*c
180
181                             Vybr_par = np.array([Rem_elements[j,3],
    Rem_elements[j,0]])
182                             if np.any(edge0[i,:] == Vybr_par[0]):
183                                 if np.any(edge0[i,:] == Vybr_par[1]):
184                                     edge0[i,0] = Vybr_par[0]
185                                     edge0[i,1] = T[3]
186                                     edge0 = np.append(edge0,[[Vybr_par[1],T
    [3]]],0)
187
188 #Presun noveho uzlu na vnitrni polomer nadoby
189                                     if Tl_nadoba == True:
190                                         p = np.array([coordinates[T[3],0],
    coordinates[T[3],1]])
191                                         c = R0/(p[0]**2+p[1]**2)**0.5
192                                         coordinates[T[3],0] = p[0]*c
193                                         coordinates[T[3],1] = p[1]*c
194
195                         for i in range(edge1.shape[0]):
196                             for k in range(3):
197                                 Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
198                                 if np.any(edge1[i,:] == Vybr_par[0]):
199                                     if np.any(edge1[i,:] == Vybr_par[1]):
200                                         edge1[i,0] = Vybr_par[0]
201                                         edge1[i,1] = T[k]
202                                         edge1 = np.append(edge1,[[Vybr_par
```

```
    [1],T[k]]],0)
203
204                         Vybr_par = np.array([Rem_elements[j,3],
    Rem_elements[j,0]])
205                         if np.any(edge1[i,:] == Vybr_par[0]):
206                             if np.any(edge1[i,:] == Vybr_par[1]):
207                                 edge1[i,0] = Vybr_par[0]
208                                 edge1[i,1] = T[3]
209                                 edge1 = np.append(edge1,[[Vybr_par[1],T
    [3]]],0)
210
211
212                     for i in range(edge2.shape[0]):
213                         for k in range(3):
214                             Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
215                             if np.any(edge2[i,:] == Vybr_par[0]):
216                                 if np.any(edge2[i,:] == Vybr_par[1]):
217                                     edge2[i,0] = Vybr_par[0]
218                                     edge2[i,1] = T[k]
219                                     edge2 = np.append(edge2,[[Vybr_par
    [1],T[k]]],0)
220
221 #Presun noveho uzlu na vnejsi polomer nadoby
222                                     if Tl_nadoba == True:
223                                         p = np.array([coordinates[T[k
    ],0],coordinates[T[k],1]])
224                                         c = R1/(p[0]**2+p[1]**2)**0.5
225                                         coordinates[T[k],0] = p[0]*c
226                                         coordinates[T[k],1] = p[1]*c
227
228                         Vybr_par = np.array([Rem_elements[j,3],
    Rem_elements[j,0]])
229                         if np.any(edge2[i,:] == Vybr_par[0]):
230                             if np.any(edge2[i,:] == Vybr_par[1]):
231                                 edge2[i,0] = Vybr_par[0]
232                                 edge2[i,1] = T[3]
233                                 edge2 = np.append(edge2,[[Vybr_par[1],T
    [3]]],0)
234
```

```
235 #Presun noveho uzlu na vnejsi polomer nadoby
236                                 if Tl_nadoba == True:
237                                     p = np.array([coordinates[T[3],0],
    coordinates[T[3],1]])
238                                     c = R1/(p[0]**2+p[1]**2)**0.5
239                                     coordinates[T[3],0] = p[0]*c
240                                     coordinates[T[3],1] = p[1]*c
241
242                         for i in range(edge3.shape[0]):
243                             for k in range(3):
244                                 Vybr_par = np.array([Rem_elements[j,k],
    Rem_elements[j,k+1]])
245                                 if np.any(edge3[i,:] == Vybr_par[0]):
246                                     if np.any(edge3[i,:] == Vybr_par[1]):
247                                         edge3[i,0] = Vybr_par[0]
248                                         edge3[i,1] = T[k]
249                                         edge3 = np.append(edge3,[[Vybr_par
    [1],T[k]]],0)
250
251                             Vybr_par = np.array([Rem_elements[j,3],
    Rem_elements[j,0]])
252                             if np.any(edge3[i,:] == Vybr_par[0]):
253                                 if np.any(edge3[i,:] == Vybr_par[1]):
254                                     edge3[i,0] = Vybr_par[0]
255                                     edge3[i,1] = T[3]
256                                     edge3 = np.append(edge3,[[Vybr_par[1],T
    [3]]],0)
257
258
259 #=====================================================================
260 #Uprava matice Hrana tak, aby v ni byly obsazeny pouze hrany sousednich
    elementu urcene k sekundarnimu presitovani
261     cnt = 0
262     while np.any(Hrana[:,-1] == 0):
263         if cnt >= Hrana.shape[0]:
264             break
265         elif Hrana[cnt,-1] == 0:
266             Hrana = np.delete(Hrana,cnt, axis=0)
267         else:
268             cnt += 1
```

```
269
270     if Hrana_memory.shape[0] <= 0:
271         pass
272     else:
273         Hrana = np.append(Hrana,Hrana_memory,0)
274
275
276     Hrana_plus = np.zeros((Hrana.shape[0]*2,5),dtype=np.int32)
277     for i in range(Hrana.shape[0]):
278         Hrana_plus[i*2] = [Hrana[i,0],Hrana[i,-1],-1,Hrana[i,3],0]
279         Hrana_plus[i*2+1] = [Hrana[i,1],Hrana[i,-1],-1,Hrana[i,3],0]
280
281     Hrana_memory = Hrana
282     Hrana = np.delete(Hrana,2,axis=1); Hrana = np.delete(Hrana,2,axis=1)
283
284     IK_B = np.zeros(12*Hrana.shape[0])
285     JK_B = np.zeros(12*Hrana.shape[0])
286     VK_B = np.zeros(12*Hrana.shape[0])
287 #   Pro kazdy stupen volnosti se musi pridat jeden blok do matice tuhosti
288     B = np.array([-0.5,-0.5,1.0])
289     Coord_shape = coordinates.shape[0]*2
290     cnt = 0
291     for i in range(Hrana.shape[0]):
292         for l in range(2):
293             if l == 0:
294                 ind = np.array([Hrana[i,0]*2, Hrana[i,1]*2, Hrana[i
    ,2]*2])
295             else:
296                 ind = np.array([Hrana[i,0]*2+1, Hrana[i,1]*2+1, Hrana[i
    ,2]*2+1])
297             for k in range(2):
298                 for j in range(3):
299                     if k == 0:
300                         IK_B[cnt] += Coord_shape
301                         JK_B[cnt] += ind[j]
302                         VK_B[cnt] += B[j]
303                         cnt += 1
304                     else:
305                         IK_B[cnt] += ind[j]
306                         JK_B[cnt] += Coord_shape
```

```
307                             VK_B[cnt] += B[j]
308                             cnt += 1
309                 Coord_shape += 1
310
311     return coordinates,elements,edge0,edge1,edge2,edge3,IK_B,JK_B,VK_B,
        Hrana_memory,Hrana_plus
```

Příloha D - CD nosič

- text diplomové práce
- Vytvořený MKP algoritmus v programovacím jazyce Python